Title: Cell-based computational modeling of vascular morphogenesis using Tissue Simulation Toolkit

Title: Cell-based computational modeling of vascular morphogenesis using *Tissue Simulation Toolkit*

Running head: Tissue Simulation Toolkit

Authors: Josephine T. Daub (1,2,3) and Roeland M.H. Merks (4,5,*)

- **Institute of Ecology and Evolution, University of Bern, Baltzerstrasse 6, 3012 Bern, Switzerland**
- **Swiss Institute of Bioinformatics, 1015 Lausanne, Switzerland**
- **Department of Ecology and Evolution, University of Lausanne, 1015 Lausanne, Switzerland**
- **Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands**
- **Mathematical Institute, University Leiden, P.O. Box 9512, 2300 RA Leiden, The Netherlands**

*** Corresponding author: merks@cwi.nl**

Abstract

Computational modeling has become a widely used tool for unraveling the mechanisms of higher-level cooperative cell behavior during vascular morphogenesis. However, experimenting with published simulation models or adding new assumptions to those models can be daunting for novice and even for experienced computational scientists. Here, we present a step-by-step, practical tutorial for building cell-based simulations of vascular morphogenesis using the Tissue Simulation Toolkit (TST). The TST is a freely available, open-source C++

library for developing simulations with the two-dimensional Cellular Potts model, a stochastic, agent-based framework to simulate collective cell behavior. We will show the basic use of the TST to simulate and experiment with published simulations of vascular network formation. Then, we will present step-by-step instructions and explanations for building a recent simulation model of tumor angiogenesis. Demonstrated mechanisms include cell-cell adhesion, chemotaxis, cell elongation, haptotaxis, and haptokinesis.

Keywords

Cellular Potts Model, agent-based modeling, Tissue Simulation Toolkit, angiogenesis, cell-based model, parameter study, quantification, Glazier-Graner-Hogeweg model

Introduction

To fully understand the mechanisms of vascular morphogenesis, it is key to unravel how higher-level cooperative cell behavior, e.g. the formation of a sprout or a vascular network, can follow from the underlying, genetically-regulated behavior of endothelial cells, and from the interactions between cells and between cells and the extracellular matrix. To this end *in silico* modeling is becoming a key tool in the study of vascular morphogenesis, in addition to *in vitro* and *in vivo* experimentation. In a cell-based *in silico* model, the researcher can test if sets of cell behavioral rules, suggested by experimental observation, suffice for reproducing aspects of the observed, higher-level cooperative cell behavior. Typically, if the researcher is successful in reproducing the higher-level cooperative cell behavior, he/she will quickly find a number of discrepancies between the experimental observations and the model outcome or find that the model does not perform well for all conditions studied. Such an observation will produce a series of model refinements and further experimental tests. Each model refinement implies an increased understanding of the mechanism.

The formation of vascular networks *in vitro* is an example problem in vascular morphogenesis that has been thoroughly studied using such an iterative computational and experimental modeling approach, which is known as *systems biology* [1]. When plated out in Matrigel or other culture gels, many endothelial cell types (e.g. human umbilical vein, endothelial cells [2] or bovine aortic endothelial cells [3]) are attracted to one another, forming vascular-like network patterns. Mathematical and computational models have suggested a range of possible mechanisms for such higher-level cooperative behavior. For example, cells may attract one another by pulling the surrounding gel towards them. This pulls neighboring cells together, and creates density gradients that cells may follow in a process called haptotaxis [4-6]. Preziosi and coworkers have proposed that

cells attract one another via chemical gradients [7-9], in a process similar to chemotactic aggregation proposed for the morphogenesis of the cellular slime mold *Dictyostelium* [10]. These initial models used continuum approximations, describing the higher-level behavior in terms of local cell concentrations.

A more recent series of models is able to capture the behavior of individual cells, an approach called cell-based modeling [11]. This makes it possible to study in a simulation model how the behavior of the individual cells is responsible for the collective. Such cell-based models, more in particular the Cellular Potts model [12], have, e.g., shown how cell shape changes the collective behavior of aggregating cells from spheroid aggregates to network patterns [13,14], studied the role of contact-inhibition in collective endothelial cell behavior [15], and showed that preferential attraction to elongated cells mediates the formation of vascular networks [16]. Such models are easily extended to model the interactions of cells and growth factors with the microenvironment, including the extracellular matrix [17-19]. A further advantage of cell-based models is that they make it possible to model how subcellular scales interact with the higher-level cell behavior, e.g., pro-angiogenic intracellular calcium signals [20] during network formation, or the mechanisms of lumen formation [21]. Also, cell-based angiogenesis models are easily integrated in larger scale models as in recent examples of vascular tumor growth [22] or age-related macular degeneration [23]. First attempts have been made to integrate the molecular, cellular, and tissue-level scale in such angiogenesis models with the goal of predicting new pharmacological, anti-angiogenic targets [20,24]. Apart from these vascular network models, a series of studies have looked at angiogenic sprouting [19,25,26].

Good reviews of cell-based models of vascular morphogenesis [27,28] and the role of modeling in vascular development [29] have been given elsewhere. In this technical book chapter, we focus on how to practically turn a hypothesis into a working, cell-based simulation model using the Cellular Potts model. A number of open source software packages are available for developing cellular Potts simulations. Which one should you use for your project? CompuCell3D [30] (available from http://www.compucell3D.org) is a large and highly user-friendly, open source software package for developing two-dimensional and three-dimensional cell-based simulation models based on the Cellular Potts model. It includes partial-differential solvers to model diffusive signals and chemoattractants, it is integrated with Systems Biology Workbench (SBW) such that the behavior of the cells can be regulated by SBML-based models of the cellular regulatory networks. Also, it is easy to use: models combining several of the increasing number of cell behaviors available in CompuCell3D can be set up in a matter of minutes using interactive 'wizards' that guide the user through the model setup, or by editing high-level, accessible XML- or Python code. Thus, if you intend to use standard, cellular Potts technology and want to develop three-dimensional simulations, CompuCell3D is your package. If you want to develop new CPM technology, e.g., experiment with the Cellular Potts algorithm [16,31], add new Hamiltonian components [19], run large parameter sweeps [15], or build novel hybrid models [32] this is certainly possible using CompuCell3D

[14,21]. CompuCell3D is well extensible, but its size and complexity can make it hard to keep the overview, and to quickly experiment with alternative algorithms, model set ups, new Hamiltonians, and so forth. If this is what you want to do then Tissue Simulation Toolkit is the package of choice. A newcomer on the "market" of Cellular Potts simulation environments is *Morpheus*[33]. It integrates two-dimensional cell-based simulations with reaction-diffusion solvers, ODE integrators in a GUI and uses an XML-based declarative model-definition language. In absence of an open source release, however, it is currently not possible to add new Hamiltonians or to flexibly build multiscale models, or to check or fine-tune the underlying simulation algorithms and numeric schemes. Although not the focus of this chapter, it is also worth mentioning examples of off-lattice cell-based modeling packages, which serves similar purposes as the Cellular Potts model: CHASTE [34], VirtualLeaf [35], and FLAME [36].

In summary, to get started with cellular Potts modeling making use of several of the available cell behaviors and model components, without dealing too much with the underlying code, using a large package like CompuCell3D or Morpheus would be the best choice. Alternatively, if for your project you need more flexibility in terms of the underlying algorithms, the Tissue Simulation Toolkit is a suitable option.

Materials

The following materials and prior knowledge are required for using and extending the code provided in this protocol.

Required software: C++ compiler and libraries

Using the TST requires basic knowledge of the use of the terminal and a C++ compiler. To acquire these skills, please refer to the appropriate books or internet tutorials.

C++ compiler: For **Windows**, unless you have Microsoft Visual Studio installed and you are familiar with its use (you will be on your own), install version gcc4.4 of the GNU MinGW compiler, from `https://code.google.com/p/psi-dev/downloads/detail?name=MinGW-gcc-4.4.0-2.7z`. Unpack the folder and rename/move it to `C:\MinGW`. Note that the required Qt library (below) will only work correctly on Windows with this particular version of MinGW.

On **MacOSX** install the XCode Development environment from the MacOSX DVDs, including the Command Line tools. You will need to specifically select these in the installer. On **Linux**, make sure that `gcc` and `g++`, and `make` are installed.

Qt Libraries: Download and install the Qt Library, version 4.8.5 from http://qt-project.org/downloads. The download page presents you with a list of Qt downloads for a range of operating systems and machine architectures. Download the Qt libraries appropriate for the operating system you are using. On **Windows** you will need the version "4.8.5 mingw4.4" (**see Note 1**) Download Qt installer for windows version "4.8.5 mingw4.4" at http://qt-project.org/downloads. The Windows Qt installer will ask for the location of your MinGW directory during installation (C:/MinGW).

LibPNG and LibZ: The TST also needs libpng and libz, both of which are often already installed on **Linux** and **MacOSX**. So for these operating systems, first attempt to compile the code (Section 2.2) before deciding to install these additional libraries. Only if necessary download sources or executables of these libraries from http://libpng.sourceforge.net and from http://www.zlib.net. On **Windows**, download the installers for these libraries from http://gnuwin32.sourceforge.net/packages/zlib.htm (complete package, except sources) and from http://gnuwin32.sourceforge.net/packages/libpng.htm (complete package, except sources). Install both libraries in C:\Program Files\GnuWin32\ so that the compiler can find them there, or alternatively edit CellularPotts2.pro if you are familiar with 'qmake'.

2.2 Source code of the Tissue Simulation Toolkit

Download the source code of the Tissue Simulation Toolkit from http://sourceforge.net/projects/tst. This tutorial is based on TST version 0.1.4.1, with download file called TST0.1.4.1.tgz.

Unpack the source code to the folder where you would like to install the TST. On Linux and MacOSX open a terminal, make a working folder, change directory to this folder ('cd [name folder]') and type:

```
> tar xzf [name download folder]/TST0.1.4.1.tgz
```

where "`>`" indicates the command prompt (i.e. start typing from 'tar'). Replace '[name download folder]' for the location of your Download folder (e.g., on MacOSX you would typically type "`tar xzf ~/Downloads/TST0.1.4.1.tgz`")

On Windows or MacOSX you can also unpack the archive by double-clicking it. Move the unpacked folder to a convenient location.

2.3 Compile the Tissue Simulation Toolkit

Windows:

Open a Qt Command prompt by choosing "Qt Command Prompt" from the "start" menu, then go to the folder where you have unpacked the source code of *TST*, e.g., (replace "[user]" for your own user name)

```
> cd c:\Documents and Settings\[user]\simulations
```

Change to the *TST* source directory.

```
> cd TST0.1.4.1\src
```

Start the compilation procedure.

```
> qmake
```

```
> mingw32-make
```

Linux and MacOSX:

Open a terminal (on MacOSX: type "Terminal" in Spotlight and press enter; Terminal is in /Applications/Utilities/).

Go to the directory where you unpacked the *Tissue Simulation Toolkit*. E.g,

```
> cd ~/simulations
```

Change to the *Tissue Simulation Toolkit* source directory.

```
> cd TST0.1.4.1/src
```

Start the compilation procedure.

Linux:

Type:

```
> qmake
```

```
> make
```

MacOSX:

```
> qmake -spec macx-g++
```

```
> make
```

Test the Tissue Simulation Toolkit

If the compilation process has proceeded well, the 'src'-folder will now contain an executable called 'vessel' (Linux and MacOSX) or 'vessel.exe' (Windows). In TST0.1.4.1, the parameter files and source files are neatly stored in different folders. Unless you are a proficient user of the TST, it is easiest to keep all the files that TST needs together in one folder. Assuming that you are currently in the src directory, type:

Windows:

```
> copy ..\data\*
```

Also retrieve the executables (e.g., vessel.exe) from the "release" folder:

```
> copy ..\release\*.exe
```

Linux and MacOSX:

```
> cp ../data/* .
```

(note the space between "*" and ".")

Next, type:

Windows:

```
> .\vessel chemotaxis.par
```

Linux and MacOSX:

```
> ./vessel chemotaxis.par
```

If all goes well, a window appears with output similar to Figure 1.

3. Methods

The *Tissue Simulation Toolkit* implements a two-dimensional cellular Potts model (CPM), a forward Euler, numerical partial-differential equation (PDE) solver suited for simple reaction-diffusion models, and a set of routines for the interaction between the PDE and CPM models. The "vessel" model that comes with the Tissue Simulation Toolkit distribution is an implementation of two previous Cellular Potts models of vasculogenesis and angiogenesis [13,15].

The cellular Potts model (CPM) represents biological cells on a regular lattice (a rectangular lattice in the TST). The lattice sites $\vec{x}$, contain integer values - called *cell ID* or *spins* – $\sigma(\vec{x}) \in \mathbb{Z}^{\{0,+\}}$, where each value of σ identifies an individual biological cell or one of the surrounding extracellular matrix materials or interstitial fluids. A (usually connected) cluster of lattice sites $\vec{x}$ containing the same cell ID $\sigma(\vec{x})$ then represents a biological cell. The CPM represents amoeboid cell motility by simulating random pseudopod extensions and retractions. To this end, the algorithm iteratively picks a random lattice site $\vec{x}$ and a random adjacent site $\vec{x}\prime$ (*i.e.*, it selects a random adjacent lattice site pair $(\vec{x}, \vec{x}\prime)$) and attempts to copy the spin $\sigma(\vec{x}\prime)$ into lattice site $\vec{x}$, as if the cell $\sigma(\vec{x}\prime)$ 'overgrows' cell $\sigma(\vec{x})$. Whether the copy attempt succeeds or not will depend on the active force the cell generates and the resulting reactive forces of the adjacent cells and materials. These forces are proportional to the change in free energy H of the system due to the copy attempt, as $F \sim \Delta H$. The copy attempts are accepted with probability $P(\Delta H) = 1$ if $\Delta H < -H_0$ (in which case the cell attempted to extend a pseudopod in the direction of least resistance), and $P(\Delta H) = \exp\left(\frac{-\Delta H + H_0}{T}\right)$ if $\Delta H \geq -H_0$, representing active, random cell motility, with T a cell motility parameter, setting the probability by which the cell will make an energetically unfavorable move, and H_0 a "dissipation energy" parameter, setting the minimal energy that must be 'overcome' in order to make a move. Typically, $H_0 = 0$ – a use for this parameter will be described in more detail below.

A typical form of a Hamiltonian describing differential cell adhesion is,

$$H = \sum_{(\vec{x},\vec{x}\prime)} J\left(\tau\left(\sigma\left(\vec{x}\right)\right), \tau(\sigma(\vec{x}\prime))\right) + \lambda_A \sum_{\sigma} \left(A(\sigma) - a(\sigma)\right)^2, \qquad (1)$$

where a copy attempt leading to a drop in the Hamiltonian will be accepted with high probability. The first term in the Hamiltonian describes cell-cell and

cell-ECM adhesion, and is a sum over all adjacent lattice pairs $(\vec{x}, \vec{x\prime})$ with the Kronecker delta term ($\delta(x, y) = 1$ if $x == y$ and $\delta(x, y) = 0$ otherwise) selecting only the cell-cell and cell-ECM interfaces. The cell-cell interfaces are associated with a positive binding energy $J(\tau_1, \tau_2)$, where high values of $J(\tau_1, \tau_2)$ give weak adhesion and lower values give stronger adhesion. To keep the number of values for J limited, they are defined between *cell types* τ, where each cell ID σ is associated with one of a small number of cell types, $\tau(\sigma)$. The second term constrains the area that the cells cover, with $A(\sigma)$ a resting area, and $a(\sigma)$ the actual, potentially compressed or expanded area of the cells

The Hamiltonian can be extended in many ways, to have the Cellular Potts model describe additional cell behaviors, such as chemotaxis, cell shape constraints, and so forth. The remainder of this chapter will describe in detail how to do so using the TST. Section 3.1 describes the basic usage of the TST, demonstrating how to run a simulation of cell sorting [12,37] and published simulations of angiogenesis and vasculogenesis [13,15]. Section 3.2 will provide some implementation details of the CPM. The main portion of this chapter is section 3.3, which demonstrate how to extend the TST and provides stepwise implementation instructions for our recent tumor angiogenesis model [19].

3.1 General usage of Tissue Simulation Toolkit

The Tissue Simulation Toolkit is a C++ code that defines the whole simulation. The compilation process (Section 2) produces an executable file that needs to be started from the command line.

The main simulation loop is defined in a main simulation file with the same name as the executable. In the standard distribution (0.1.4.1) of the TST two such main files are given as examples. We will illustrate the use of the Tissue Simulation Toolkit with a simulation of differential-adhesion driven cell sorting [12,37] according to the Hamiltonian defined above with two cell types.

1. Compile the cell sorting simulation, which is defined in 'sorting.cpp'. To tell the compiler to compile "sorting" instead of "vessel", open the file "CellularPotts2.pro" in a text editor. Navigate to the line that starts with "TARGET =" (line 13 in TST0.1.4.1), and change it to:

   ```
   TARGET = sorting
   ```

   and save the "CellularPotts2.pro" file.

2. Recompile TST following the steps listed in Section 2.3.

3. Start the program by typing the executable name, followed by the name of the parameter file
   `./sorting sorting.par`
   into a terminal or DOS-box. Replace "./" for ".\" on Windows. A window will appear with the initial cluster and a mixed pattern of cells will appear.

4. Now let's try to change the parameters in order to get the cells to stick together and sort out, or to make one type of cells to engulf the other type. To do so, first make a copy of the file J.dat, by typing:

   ```
   > copy Jnoadhesion.dat myJ.dat
   ```
   or (for Linux)

   ```
   > cp Jnoadhesion.dat myJ.dat
   ```
   into a terminal or DOS-box. Then open the file myJ.dat. It will look something like this:

   ```
   3
   0
   20 40
   20 40 40
   ```

   The "J-file" describes the adhesion energies as a diagonally symmetric matrix J; the first line gives the number of cell types, 3. That is two cell types (red, yellow) plus one for the ECM (white).

   The second line gives $J(M, M)$, the adhesion energy between "medium cells". Since we only have one medium "cell", there are not boundaries between cells of type "medium"; thus $J(M, M) = 0$ by definition. The next line describes the adhesion energies between cell type 1 and the other cell types and the medium. The first number is the adhesion energy between cell 1 and the medium, $J(M, 1) = J(1, M)$, the second number the adhesion energy, $J(1, 1)$ of cell 1 to its own type. Similarly, the fourth line lists ($J(0, 2)$, $J(1, 2)$, and $J(2, 2)$.

5. To make the TST look for your new "J-file", open file sorting.par in a text editor (e.g. notepad) and change the line:
   `Jtable = Jnoadhesion.dat`

   to
   `Jtable = myJ.dat`

   See also Step 7 for more information on the format of the parameter files.

6. Experiment with different values of J. For example, to have the cells of type 1 stick together, set the parameters such that $J(1, 1) < 2J(M, 1)$. In this way a boundary between two cells of type 1 is energetically more favorable than two cell-ECM boundaries of the same length. We leave the other cases as an exercise to the reader.

7. All the other parameters are defined in the parameter file with extension '.par'. The format of this file is as follows. All lines starting with '#' are treated as comments and will be ignored. Parameter values are assigned by giving the parameter name, a mandatory space, then the '=' sign,

followed by another space, and the parameter value. For example, the following line sets the value of λ in the Hamiltonian:

```
lambda = 50
```

Similarly, T defines the value of the cell motility parameter, and `target_area` gives the value of the target area, A.

8. The parameter file gives the value of additional parameters not mentioned in the CPM model definition above – an overview is given in Table 1.

We will next describe the basic structure and usage of two previously published models of angiogenesis and vasculogenesis [13,15]. Both of these models assume that endothelial cells secrete a diffusive chemoattractant (e.g. VEGF), which is simulated using a continuum, partial-differential equation model. A PDE model is coupled to the CPM to describe the chemoattractant, c; it is secreted by the endothelial cells, it diffuses, and it is degraded in the extracellular matrix,

$$\frac{\partial c}{\partial t} = D\nabla^2 c + \alpha(1 - \delta(\sigma(\vec{x}), 0)) - \epsilon\delta(\sigma(\vec{x}), 0)c \tag{2}$$

with $\delta(\sigma(\vec{x}, 0)) = 1$ in the medium, and $\delta(\sigma(\vec{x}, 0)) = 0$ at lattice sites covered by cells. We implement chemotaxis by assuming that cells preferentially extend pseudopods in the direction of chemoattractant gradients [38]. The algorithm allows for an extra energy drop at the time of copying:

$$\Delta H_{\text{chemotaxis}} = -\mu(c(\vec{x}\prime) - c(\vec{x})) \tag{3}$$

where $(\vec{x}, \vec{x}\prime)$ is an adjacent lattice site pair.

9. The program vessel.cpp implements the angiogenesis and vasculogenesis simulations. If all went well, you have compiled this program in Section 2 and an executable "vessel" or "vessel.exe" (Windows) is in your working folder. If not, follow steps 1 and 2 of Section 3.1 (i.e., make sure the CellularPotts.pro file reads `TARGET = vessel`).

10. Start up a Cellular Potts simulation with the parameter file chemotaxis.par, by typing on the command line:
`vessel chemotaxis.par`

11. Reduce the display frequency to observe the long time behavior of the simulation. Edit the parameter file, setting `storage_stride = 100`.

As in the PDE model by Gamba *et al.* [7], the cells secrete a chemoattractant (shown in grey and with the green concentration isolines), and move towards higher concentrations of the attractant. The endothelial cells should form rounded aggregates as in Figure 1.

12. Experiment with the parameters, including the decay rate of the chemoattractant (`decay_rate`), the diffusion coefficient (`diff_coeff`), and the number and size of the cells (`n_init_cells`, `target_area`). Stop the simulation by pressing "CTRL-C" in the Terminal or closing the graphics window. Edit the parameter file as described in Step 5 above. Then restart the simulation as described in Step 10. A recommended exercise is to study what determines the size of the cell clusters. **See also Note 2 for numerical instability issues.**

13. With additional assumptions, including elongated cells [13] or contact inhibition [15] this chemotactic mechanism suffices for producing vascular-like structures. To mimic cell elongation due to cytoskeletal remodeling we add a cell length constraint to the Hamiltonian:

$$H\prime = H + \lambda_L \sum_{\sigma} \left(l_\sigma - L_\sigma\right)^2 , \tag{4}$$

where l_σ is the length of cell along its longest axis, L_σ its target length, and λ_L is the strength of the length constraint. Assuming that cells are ellipses, we can derive their length from the largest eigenvalue of their inertia tensor. The length constraint could cause cells to split into disconnected patches. We prevent this artifact by introducing a connectivity constraint, which reflects the physical continuity and cohesion of the actual cell. This connectivity constraint sets the "dissipation energy" parameter, H_0, to a large positive value for copy updates that would locally break the connectivity of the cell (see [13] for detail).

14. Start from a clean parameter file, by copying `chemotaxis.par` and editing the copy. Experiment with the value of `target_length` (L_σ); a good value to start with would be `target_length = 60` ($L = 120$ μm, if `dx=2.0e-6`).

15. Experiment with small numbers of cells, say `n_init_cells = 10`. What happens to the polygonal pattern?

16. To increase the field size, increase the field size using the parameters `sizex` and `sizey`, *e.g.*, set both values to 500. A useful parameter to change is also `subfield`; if you set this to `subfield = 1.5` the cells will be distributed within a restricted space in the center of the field so the network has sufficient space to expand.

An alternative mechanism for network formation and sprouting is contact inhibition of chemotaxis. In this mechanism, we assume that only the cell-ECM interfaces sense the chemoattractant, a mechanism that phosphorylation of VEGFR-2 receptors by vascular-endothelial cadherin might mediate (see [15] for detail).

17. Start with a clean copy of `chemotaxis.par`. Make sure that the cell shape is constrained (`lambda2 = 5.0`). Switch on the contact-inhibition

mechanism: `vecadherinknockout = false`. Save the parameter file, and restart the simulation. If all proceeds well, the cells will form more elongated aggregates.

18. To see a network of cells, increase the field size and increase the number of cells accordingly. Suitable values are: `sizex = 500`, `sizey = 500`, `n_init_cells = 500`, `subfield = 1.5`. To see a network develop more quickly, within the standard field of 200×200, reduce the diffusion length (width of the chemoattractant gradients), e.g., by increasing the secretion and degradation rates of the chemoattractant, `decay_rate = 1e-3`, `secr_rate = 1e-3`, the values used in [15], and use a larger number of smaller cells, `target_area = 50`, and `n_init_cells = 300` are suitable values. **See Note 3.**

19. Next initialize the simulation with a spheroid of cells. To do so, we make use of TST's cell division algorithm. Initialize the simulation with a single, large cell. Start with the parameter settings suggested in Step 18. Set `n_init_cells = 1` and `size_init_cells = 50`. In the TST, cells are initialized with an Eden growth [39] algorithm that produces disc-shaped growth patterns. `size_init_cells` gives the number of Eden growth cycles. Cleave the cell for a number of times, e.g., by setting `divisions = 7` to create 128 initial cells. Start up the simulation.

20. In Step 19, the initial spheroid quickly expands to satisfy the cells' area constraints. To ensure that this process does not interfere with the observed pattern formation, it is possible to run only the CPM steps for a couple of Monte Carlo Steps (MCS) such that the initial cells 'relax' to their ideal size before starting to chemotact. Set parameter `relaxation = 200`, or the number of MCS that you prefer.

The above simulations use "extension-retraction" chemotaxis [15], in which chemoattractant gradients both guide pseudopod extensions (i.e. spin copy from site covered by cell to adjacent cell or medium cell) and pseudopod retractions (*i.e.*, copy of medium site into site covered by cell). In the alternative, "extension only" chemotaxis mechanism only pseudopod extensions are guided by the chemoattractant gradients [15].

21. In the parameter file written for steps 17-20, set `extension_only = true`. Rerun the simulation. The spheroid will not sprout.

22. Random cell motility can drive sprouting with extension-only chemotaxis due to a curvature effect (see [15] for detail). Increase the cellular temperature to `T = 200`, and rerun the simulation.

An interesting mechanism is that of "passive cell shape changes" [40]. In this mechanisms cells elongate as they are 'trapped' by steep chemoattractant gradients they produce themselves. Such steep gradients might, e.g., be due to binding of the secreted morphogens to matrix components [17,18].

23. Start from a clean copy of chemotaxis.par, and release the shape constraint from the endothelial cells, by setting `lambda2 = 0.0`, and run a new simulation. To speed up the simulation reduce the target areas, increasing the number of cells. Run the simulation; it should produce rounded clusters.

24. Reduce the width of the chemoattractant gradients, e.g., by reducing the diffusing rate of the chemoattractant to `diff_coeff = 1e-14`. Run the simulation; it should now produced network-like patterns.

3.3 Structure of Tissue Simulation Toolkit programs

A basic model, using only the standard model components already implemented in the TST is implemented using a main C++ file defining the simulation. The easiest way to build your own simulation is to edit one of the example models "sorting.cpp" or "vessel.cpp". The main C++ file contains three main code blocks.

```
// Header files included here

INIT {
   // define initial condition
}

TIMESTEP {
  // call a Monte Carlo step here,
  /// and call the PDE integrator if necessary
  /// Also call all routines that need to be
  /// performed after one or several
  /// Monte Carlo steps (e.g. visualization)
}

int PDE::MapColour(double val) {
  // optional: used for PDE visualization
}

int main(int argc, char *argv[]) {
  // Initialize application
  // Will rarely need to be changed
  return 0;
}
```

The INIT code block defines the initial condition of the simulation. The TIMESTEP code calls one Monte Carlo Step, it calls the partial-differential equation solver, and it performs routines that need to be taken care of after each or after a number of Monte Carlo steps, including the visualization. The `main` function initializes the application, and will rarely need to be changed. An

additional number of optional functions can be implemented here, including the `PDE::MapColour` functions that is used for visualization of PDEs.

In this subsection we will briefly document the code of the `vessel.cpp` model. For a full class structure and class documentation, we refer to the HTML documentation provided with the TST. See TST0.1.4.1/doc/html/index.html.

3.3.1. Code block INIT

Here we briefly document the code block INIT of 'vessel.cpp'.

```
INIT {

  try {
```

The 'try' and 'catch' statements (below) are an error handling mechanism; it is not necessary to understand it in detail as long as you do not change the structure.

The `CPM->` construction indicates that these are member functions of the class `CellularPotts`; i.e., they read out or change the Cellular Potts plane. To find more information on the use of these functions in the HTML documentation, click "Class Hierarchy", then "CellularPotts" and look for the name of the function. The `par.` construction indicates that we are taking the value of a parameter defined in the parameter file of the same name.

```
  // Define initial distribution of cells
  /// CPM->GrowInCells(par.n_init_cells,par.size_init_cells,
  ///    par.subfield);
```

`CPM->GrowInCells` sets `n_init_cells` pixels to a spin of increasing value, and expands them by applying an Eden growth algorithm [39] for `size_init_cells` time steps. This will produce approximately disc-shaped initial cells. The cells are distributed over an area of size sizex/subfield by sizey/subfield centered within the CPM plane.

`GrowInCells` will simply draw cells on the CPM plane. As a next step, we will need to construct an object of class `Cell` for each of the patches. If you want to construct your own initial condition, simply copy `GrowInCells` to your own function, and have it draw cells where you like them. As long as you call `ConstructInitCells` afterwards this will create a viable initial condition.

```
CPM->ConstructInitCells(*this);
```

Here is a call to the cell division algorithm that Step 19 used to generate an initial spheroid of cells. Section 3.4.5 illustrates how to call the `DivideCells` algorithm from within a running simulation to have your cells proliferate. Briefly, a variant of the `DivideCells` function takes a Boolean array `which_cells` that marks the cells for division.

```
  // If we have only one big cell and divide it a few times
  // we start with a nice initial clump of cells.
  //
  // The behavior can be changed in the parameter file.
  for (int i=0;i<par.divisions;i++) {
    CPM->DivideCells();
  }
```

This is the closing statement of the try/catch error handling construct.

```
  } catch(const char* error) {
     std::cerr << "Caught exception\n";
     std::cerr << error << "\n";
     std::exit(1);
  }
```

3.3.2. Code block TIMESTEP

The code block `TIMESTEP` implements one time step of the simulation, and is called from the main loop. It typically runs one MCS, runs one or more partial-differential equation steps, and calls the output functions.

```
TIMESTEP {

  try {
```

'i' is a counter that we can increment each time the TIMESTEP is run.

```
  static int i=0;
```

The object 'dish' of class 'Dish' contains everything we need to run a simulation, i.e., the cellular Potts algorithm, a list of cells, the PDE solver, and so forth.

```
  static Dish *dish=new Dish();
```

The helper class `Info` collects information from the cells, e.g., by clicking on them interactively.

```
  static Info *info=new Info(*dish, *this);
```

Run the partial-differential equation solver for `pde_its` time steps, but only after `relaxation` time steps.

```
  if (i>=par.relaxation) {

    for (int r=0;r<par.pde_its;r++) {
```

The object `PDEfield` of class `PDE` contains a PDE field and the numerical solver. Function `Secrete` implements the non-diffusive sections of the PDE. The name only covers part of the function, because in this example it implements both secretion and degradation, and it could implement additional PDE terms. Dish->CPM is given as an argument to apply some terms only within the cells or only in the medium, cf. the Kronecker delta terms in the model equations.

```
      dish->PDEfield->Secrete(dish->CPM);
```

Call 1 iteration of the finite-difference diffusion algorithm.

```
      dish->PDEfield->Diffuse(1);

    }

  }
```

Call one MCS of the Cellular Potts model. The PDE field is passed as an argument for the chemotaxis algorithm.

```
dish->CPM->AmoebaeMove(dish->PDEfield);
```

Here we call the visualization code. The first block writes to the graphics window, every other `storage_stride` MCS.

```
  if ((par.graphics || par.store) && !(i%par.storage_stride)) {
```

`BeginScene` prepares the graphics library for image writing.

```
   BeginScene();
```

Plot the PDE field. `this` indicates that the current graphics window is used. Always write 'this' here. The second argument gives the number of the layer that will be plotted.

```
  dish->PDEfield->Plot(this,0);
```

Plot the CPM model output over the PDE field. The medium is left transparent to leave the PDE field visible. If you do not draw a PDE field, call `ClearImage()` first.

```
  dish->Plot(this);
```

This is a second visualization of the PDE field, using contour lines. The first two arguments again indicate the graphics window and the number of the field. The third argument, '7', gives the color, as defined in `default.ctb`; **see Note 4.**

```
   if (i>=par.relaxation)
     dish->PDEfield->ContourPlot(this,0,7);
```

`EndScene` flushes the data to the graphics window.

```
EndScene();
```

The next block of code writes the graphics to a PNG file.

```
if (par.store) {

    char fname[200];
    sprintf(fname,"\%s/extend\%05d.png",par.datadir,i);

    Write(fname);

    }
  }
```

Increment the time counter.

```
  i++;

 } catch(const char* error) {

  cerr << "Caught exception\n";
  std::cerr << error << "\n";
  exit(1);

  }

}
```

3.3.3. Remaining code blocks in vessel.cpp: PDE::Secrete, PDE::MapColour, main

Two remaining code blocks in vessel.cpp contain part of the implementation of the angiogenesis model. PDE::Secrete, a member function of class PDE, defines most of the partial-differential equation component of the model, except the diffusion operator.

```
void PDE::Secrete(CellularPotts *cpm) {
```

Loop over the whole lattice.

```
 for (int x=0;x<sizex;x++)

  for (int y=0;y<sizey;y++) {
```

This term is only executed inside the cells, *i.e.*, it reads $\alpha(1-\delta(\sigma(\vec{x})))$. Read `sigma[i][x][y]` as $c_i(\vec{x})$, with c_i the concentration in ith PDE layer.

```
   // inside cells

   if (cpm->Sigma(x,y)) {

       sigma[0][x][y]+=par.secr_rate[0]*par.dt;

   } else {
```

This term is only executed outside the cells, i.e., it reads $-\epsilon c_0(\vec{x})\delta(\sigma(\vec{x}))$.

```
     // outside cells

     sigma[0][x][y]-=par.decay_rate[0]*par.dt*sigma[0][x][y];
   }

  }

}
```

Function `PDE::MapColour` maps a chemical value c to a grey value g in the range $g \in [0, 100]$ as $g = c/(1+c) * 100$. Finally function `main` initializes the application and will usually remain unchanged.

3.3.4. Implementing the Hamiltonian function: a brief overview over class `CellularPotts`

The Cellular Potts algorithm is implemented in class `CellularPotts`; see file `ca.cpp`. In order to add your own components to the Hamiltonian, the two key functions to look for are `CellularPotts::AmoebaeMove`, that implements one Monte Carlo Step, and `CellularPotts::DeltaH` that is responsible for calculating the energy change associated with a potential copy step.

`AmoebaeMove` repeatedly picks a random lattice site with coordinates `x, y` and a random, adjacent lattice site `xp, yp`, and attempts to copy the spin $\sigma(\vec{x})$, sigma[xp][yp], into lattice site $\vec{x}$, `sigma[x][y]`. Function `DeltaH` calculates the energy change, ΔH, that would result from this potential update. Function `ConvertSpin` actually performs the copy with probability $P(\Delta H) = \{1, \Delta H + H_0 \leq 0; \exp((-\Delta H + H_0)/T), \Delta H + H_0 > 0\}$.

To change the Hamiltonian, change `DeltaH`. The standard implementation includes (in that order) cell adhesion, an area constraint, the chemotaxis algorithm, and the length constraint. These components are switched on and off depending on the parameter values. How to extend the Hamiltonian with additional components, e.g., haptotaxis and haptokinesis, is illustrated in the next Section.

3.4 Implementation of an ECM-guided angiogenesis model using the TST

In this section we will build up a model of extra-cellular matrix (ECM)-guided tumor-induced angiogenesis as described previously [19]. We will add multiple chemical fields and let concentrations in one field influence the secretion or decay of chemicals in another field. The includes the following rules: 1) Tumors secrete the growth factor VEGF resulting in a VEGF gradient [41]; 2) VEGF stimulates the secretion of diffusive matrix metalloproteinases (MMPs) by endothelial cells (ECs), and 3) MMPs break down ECM components near the cell surface [42,43]; 4) ECs move chemotactically along VEGF gradients [44,45], and they 5) migrate toward higher ECM densities (haptotaxis) [46,47]. 6) Cell speed and spreading are optimal at intermediate ECM densities [48,49], and 7) cells proliferate if a large part of their surface is in contact with the ECM [50].

We will describe how to implement these steps, specifically how to change the Hamiltonian to model haptokinesis and haptotaxis. We will follow the line of the paper by Daub and Merks (2013) [19] and add functionality step by step, resulting in simulations similar to those of Figure 3 in the paper by Daub et al. [19].

25. Set up a new working directory: Copy or rename the original `tst` folder to `ecm`.

26. We will use the `vessel.cpp` and `default.par` files as templates, which we will change for our model. Therefore, copy and rename `vessel.cpp` to `ecm.cpp` and copy `default.par` in the `data` directory to `ecm.par`.

27. Open `CellularPotts2.pro` and change the target to `ecm`.

28. Make sure that everything is working, by compiling and running the program.

3.4.1 Dish setup: cells behind a vessel wall

We will first set up the initial situation: ECs are placed behind a vessel wall at the bottom of a rectangular dish. There is a small gap in the vessel wall, through which the cells can migrate (Figure 2A).

29. Introduce two new parameters, `gapx` and `gapy` to set up the vessel wall in the dish and place the cells behind it. The parameter `gapx` defines the width of the gap and `gapy` the vertical placement of the vessel in the dish. In `parameter.cpp` (in the `src` directory) edit the following lines of code (**see Note 5**):

```
Parameter::Parameter() {

   (...)

  sizex = 200;
  sizey = 200;
  gapx = 0;
  gapy = 0;

  (...)
}

void Parameter::Read(const char *filename) {

 (...)
 sizex = igetpar(fp,  "sizex", 200, true);
 sizey = igetpar(fp,  "sizey", 200, true);
 gapx = igetpar(fp, "gapx", 0, true);
 gapy = igetpar(fp, "gapy", 0, true);

 (...)

}
```

30. Add the new parameters also to the header file `parameter.h`:

```
class Parameter {


public:
```

```
  (...)

  int sizex;
  int sizey;
  int gapx;
  int gapy;

  (...)

}
```

31. Create the vessel wall during the initialization of the dish. In `ca.cpp` add the following lines of code to the initialization functions of the `CellularPotts` object:

```
CellularPotts::CellularPotts(vector<Cell> *cells,
                             const int sx, const int sy) {
  (...)

  // fill borders with special border state
  for (int x=0;x<sizex;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
  }
  for (int y=0;y<sizey;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
  }

  // add vessel wall with gap
  if (par.gapy) {
    for (int x=0;x<(sizex-par.gapx)/2;x++){
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
    for (int x=(sizex+par.gapx)/2;x<sizex;x++){
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
  }

  (...)
}
```

```
CellularPotts::CellularPotts(void) {

  (...)

  // fill borders with special border state
  for (int x=0;x<sizex;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
  }
  for (int y=0;y<sizey;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
  }

  // add vessel wall with gap
  if (par.gapy) {
    for (int x=0;x<(sizex-par.gapx)/2;x++){
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
    for (int x=(sizex+par.gapx)/2;x<sizex;x++){
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
  }

  (...)
}
```

32. We will create a new function `IsWall` to test whether a grid point is part of the vessel wall. Add this function to `ca.cpp`:

```
int CellularPotts::GrowInCells(int n_cells, int cell_size,
      int sx, int sy, int offset_x, int offset_y) {

    (...)

}

// Check whether (x,y) is part of the vessel wall

bool CellularPotts::IsWall(int x, int y) {

  if (!par.gapy)
```

```
   return false;
  else
    if ((y==par.gapy || y==par.gapy+1) &&
        (x<(par.sizex-par.gapx)/2 ||
          x>=(par.sizex+par.gapx)/2))
      return true;
else
  return false;
}
```

33. Add the public function `IsWall` also to `ca.h`:

```
class CellularPotts {
(...)
public:
  int GrowInCells(Int n_cells, int cell_size, int sx, int sy,
    int offset_x, int offset_y);
  // Returns true if lattice site (x,y) is part of vessel wall

  bool IsWall(int x, int y);
  (...)
};
```

34. Change both versions of the function `GrowInCells` in `ca.cpp`, in order to build the vessel wall and place cells behind it:

```
int CellularPotts::GrowInCells(int n_cells, int cell_size,
                               double subfield) {
  int sx, sy, offset_x, offset_y;

  if (!par.gapy){
    sx = (int)((sizex-2)/subfield);
    sy = (int)((sizey-2)/subfield);

    offset_x = (sizex-2-sx)/2;
    offset_y = (sizey-2-sy)/2;
  } else {
    // If there is a vessel wall in the dish,
    // cells are grown behind this wall
    sx = (int)((sizex-2)/subfield);
    sy = sizey-3-par.gapy;

    offset_x = (sizex-2-sx)/2;
    offset_y = par.gapy + 1;
  }
```

```
  if (n_cells==1) {
    (...)
  }
}


int CellularPotts::GrowInCells(int n_cells, int cell_size,
                               int sx, int sy, int offset_x,
                               int offset_y) {
  (...)

  // copy sigma to new_sigma, but do not touch the border!
  {for (int x=1;x<sizex-1;x++) {
     for (int y=1;y<sizey-1;y++) {
       if (!IsWall(x,y))
         sigma[x][y]=new_sigma[x][y];
     }
   }

  (...)

  return cellnum;
}
```

35. Add the following lines to the function `ThrowInCells` in `ca.cpp`:

```
int CellularPotts::ThrowInCells(int n,int cellsize) {

(...)

  // repair borders
  // fill borders with special border state
  for (int x=0;x<sizex-1;x++) {
    sigma[x][0]=-1;
    sigma[x][sizey-1]=-1;
  }
  for (int y=0;y<sizey-1;y++) {
    sigma[0][y]=-1;
    sigma[sizex-1][y]=-1;
  }

  // add a vessel wall with gap
  if (par.gapy) {
    for (int x=0;x<(sizex-par.gapx)/2;x++){
```

```
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
    for (int x=(sizex+par.gapx)/2;x<sizex;x++){
      sigma[x][par.gapy]=-1;
      sigma[x][par.gapy+1]=-1;
    }
  }
  (...)
}
```

36. We have to make sure that cells cannot move through the wall. Make a small change in the `AmoebaeMove` function in `ca.cpp`:

```
int CellularPotts::AmoebaeMove(PDE *PDEfield)
{
  (...)

    // test for border state (relevant only if we do not use
    // periodic boundaries)
    if (kp!=-1) {

      // added k!=-1 to avoid copying into the vessel wall
      if (k!=-1 && k!=kp) {

         (...)

      }
    }
  }
  return SumDH;
}
```

37. Set the parameters in the parameter file `ecm.par` in the `data` folder. We will mostly follow the parameter settings in Table 1 of [19], except for chemotaxis, which we set to `chemotaxis = 0` for now.

```
# Cellular Potts parameters
T = 100
target_area = 50
target_length = 15
lambda = 25
lambda2 = 25
Jtable = J.dat
conn_diss = 2000
```

```
vecadherinknockout = false
chemotaxis = 0
border_energy = 100
extensiononly = true

neighbours = 3
periodic_boundaries = false

# PDE parameters
n_chem = 1
diff_coeff = 1e-13
decay_rate = 1.8e-4
secr_rate = 1.8e-4
saturation = 0
dt = 2.0
dx = 2.0e-6
pde_its = 15

# initial conditions
n_init_cells = 125
size_init_cells = 13

sizex = 250
sizey = 350
gapx = 25
gapy = 320

divisions = 0
mcs = 10000
rseed = -1
subfield = 1.05
relaxation = 0

# output
storage_stride = 10
graphics = true
store = false
datadir = data_film
```

38. Change the energy settings in `J.dat` in the `data` folder to:

```
2
0
25 40
```

39. Compile and run the application, making sure that you save all files that

have been changed in this section. You will now see the ECs placed behind the vessel wall and migrating through the gap in the vessel wall. You can change the `gapx` parameter to see the effect of the gap size.

3.4.2. Dish setup: Static VEGF gradient

At the top of the dish a tumor secretes VEGF. The ECs can migrate through the gap in the parent vessel wall and move chemotactically up the VEGF gradient toward the tumor. In our model we assume a planar steady-state VEGF gradient (Figure 2B) given by:

$$c_v(x_2) = c_v(0)e^{-x_2/\lambda} \text{ with } \lambda = \sqrt{D/\epsilon} \tag{5}$$

where $c_v(x_2)$ is the VEGF concentration at distance y from the tumor, and D and ϵ are the diffusion coefficient and decay rate of VEGF. We will initialize the gradient once and it will not further change during the simulation.

40. Set up the VEGF gradient using the diffusion and degradation parameters (`diff_coeff` and `decay_rate`) as well as the VEGF concentration at the tumor. We create a new parameter `init_conc`, to set this VEGF concentration in the parameter file. As this parameter will also be used to initialize other PDE layers, it will be a list of values. We will also create a new parameter `chem_type`, which defines the chemical type for each layer (VEGF, ECM, MMP, or others). Add the following lines of code to `parameter.cpp`:

```
Parameter::Parameter() {
  (...)
  n_chem = 1;
  chem_type = new double[1];
  chem_type[0] = 0.;
  init_conc = new double[1];
  init_conc[0] = 0.;
  (...)
}

void Parameter::CleanUp(void) {
  if (Jtable)
     free(Jtable);
  if (chem_type)
    free(chem_type);
  if (init_conc)
    free(init_conc);
  (...)
}
```

```
void Parameter::Read(const char *filename) {
  (...)
  n_chem = igetpar(fp, "n_chem", 1, true);
  chem_type = dgetparlist(fp, "chem_type", n_chem, true);
  init_conc = dgetparlist(fp, "init_conc", n_chem, true);
  (...)
}

void Parameter::Write(ostream &os) const {
  (...)
  os << "n_chem = " << n_chem << endl;
  os << "chem_type = " << chem_type[0] << endl;
  os << "init_conc = " << init_conc[0] << endl;
  (...)
}
```

41. Add the parameters to `parameter.h`:

```
class Parameter {

 public:
  (...)
  int n_chem;
  double * chem_type;
  double * init_conc;
  (...)
};
```

42. Define the first PDE layer for the VEGF concentrations in the dish. As our model will have different PDE layers, each with a different chemical, we will annotate each layer which makes it easier to read and to change the code. In `ecm.cpp` add the VEGF chemical and layer at the beginning of the file.

```
// Chemicals in ecm application
const int VEGF = 0;

// The layers in ecm application
int L_VEGF = -1;

using namespace std;
```

43. Add the following code to the `INIT` block in `ecm.cpp`, to give each layer the proper name. In the parameter file we will define the chemical type

for each layer. This way you do not necessarily need to define all 3 layers and you are free to change the order.

```
INIT {
    (...)

    // The behavior can be changed in the parameter file.
    for (int i=0;i<par.divisions;i++) {
      CPM->DivideCells();
    }

    // Assign layer indices to specific layer variables
    for (int l=0;l<PDEfield->Layers();l++){
      switch ((int) par.chem_type[l]) {
        case VEGF: L_VEGF=l;
        break;
      }
    }

    (...)
}
```

44. Add the code to initialize the VEGF PDE field. By default, other layers will be initialized with a uniform chemical density (`init_conc`). Later on we will add code to set up the ECM field

```
INIT {

  try {

   (...)

    // Assign layer indices to specific layer variables
   (...)

    // Initialisation of PDE fields
    for (int x=0;x<par.sizex;x++)
    for (int y=0;y<par.sizey;y++)
      for (int l=0;l<PDEfield->Layers();l++){

        if (l==L_VEGF) {

          double D = par.diff_coeff[l];
          double d = par.decay_rate[l];
          double b, conc;
```

```
          b = -1/sqrt(D/d);
          // Calculate gradient, when concentration at 'top'
          // of dish is given
          conc = par.init_conc[l]*exp(b*y*par.dx);
          PDEfield->setValue(l,x,y,conc);

        } else {
         /* ======================================================
         /  OTHER LAYERS: UNIFORM DENSITY
         ======================================================*/
          PDEfield->setValue(l,x,y,par.init_conc[l]);
        }
      }

  (...)

}
```

45. As the VEGF gradient is static throughout the simulation, we will change the `Diffuse` function in order to skip layers that will not diffuse during the simulation. In `pde.cpp` add a new version of the `Diffuse` function:

```
void PDE::Diffuse(int repeat) {
  (...)
}

// Only diffuse layers not in skiplist
void PDE::Diffuse(int repeat, const int * skiplist, int nrskips) {

  const double dt=par.dt;
  const double dx2=par.dx*par.dx;
  double D;

  for (int r=0;r<repeat;r++) {
    //NoFluxBoundaries();
    if (par.periodic_boundaries) {
      PeriodicBoundaries();
    } else {
      AbsorbingBoundaries();
      //NoFluxBoundaries();
    }

    for (int l=0;l<layers;l++) {
      D = par.diff_coeff[l];
```

```
      for (int i=0;i<nrskips;i++)
        if (l==skiplist[i]) {
          D=0;
          break;
        }
      for (int x=1;x<sizex-1;x++)
        for (int y=1;y<sizey-1;y++) {
          if (D) {
            double sum=0.;
            sum+=sigma[l][x+1][y];
            sum+=sigma[l][x-1][y];
            sum+=sigma[l][x][y+1];
            sum+=sigma[l][x][y-1];

            sum-=4*sigma[l][x][y];
            alt_sigma[l][x][y]=sigma[l][x][y]+sum*dt*D/dx2;
          }
          else
            alt_sigma[l][x][y]=sigma[l][x][y];
      }
    }
    double ***tmp;
    tmp=sigma;
    sigma=alt_sigma;
    alt_sigma=tmp;

    thetime+=dt;
  }

}
```

46. In `pde.h` add the new `Diffuse` function:

```
class PDE {
  (...)
  public:
    (...)
    void Diffuse(int repeat);
    void Diffuse(int repeat, const int * skiplist, int nrskips);
    (...)
};
```

47. In the `TIMESTEP` block in `ecm.cpp` we add the new `Diffuse` functionality and disable the call to `Secrete` for now:

```
TIMESTEP {

  try {

    (...)
    // Put layers with diffusion coefficient>0
    // but who do not diffuse during the simulation
    // in Skiplist
    static int SkipList[1] = {L_VEGF};
    if (i>=par.relaxation) {
      for (int r=0;r<par.pde_its;r++) {
        //dish->PDEfield->Secrete(dish->CPM);
        dish->PDEfield->Diffuse(1, SkipList, sizeof(SkipList));
      }
    }

    dish->CPM->AmoebaeMove(dish->PDEfield);
  }
}
```

48. Set the VEGF parameters in the parameter file `ecm.par` in the `data` folder.

```
# Cellular Potts parameters
chemotaxis = 6000

# PDE parameters
chem_type = 0
init_conc = 0.87
diff_coeff = 6e-11
decay_rate = 1e-3
secr_rate = 0
```

49. Compile and run the application. You will now see the vessel growing slowly towards the tumor at the top of the dish. You can see the effect of the `chemotaxis` parameter by setting it to different values. Setting it to zero will result in the first panel of Figure 3 in [19]. Increasing the chemotaxis coefficent will increase the speed with which the vessel grows towards the tumor. If you set it very high (for example at 50000) cells will split off, leading to simulation artifacts. As the simulation is rather time consuming, you could change the output parameters to save the graphic to a file every 1000 MCS. Make sure you have created a folder named `data_film` in your data directory. **See Note 6.**

```
# output
storage_stride = 1000
```

```
graphics = true
store = true
datadir = data_film
```

3.4.3. Adding interaction with the extra-cellular matrix: haptokinesis

In the coming sections we will add the effect of interactions between ECs and the ECM to the model. We will start with including `haptokinesis` in our simulation program, the phenomenon that cells have optimal motility at intermediate ECM densities.

50. Change and add parameters. Add a parameter `haptokinesis` to define the strength of haptokinesis per layer, and change the `chemotaxis` parameter, to set this parameter per layer as well. Furthermore, introduce two new parameters `min_conc` and `max_conc`, to set the minimal and maximal concentration per pde layer. For the graphics we will define new parameters to assign colors per layer: `cont_color` for contour lines and `grad_color` for the gradient. In `parameter.cpp` change the `chemotaxis` parameter to an array and add the new parameters (**See Note 7 and Note 8**):

```
Parameter::Parameter() {

  secr_rate = new double[1];
  secr_rate[0] = 1.8e-4;
  chemotaxis = new double[1];
  chemotaxis[0] = 1000.;
  haptokinesis = new double[1];
  haptokinesis[0] = 10.;
  max_conc = new double[1];
  max_conc[0] = 99;
  min_conc = new double[1];
  min_conc = 0;
  cont_color = new double[1];
  cont_color[0] = 7.;
  grad_color = new double[1];
  grad_color[0] = 1.;
}

void Parameter::CleanUp(void) {
  (...)
  if (secr_rate)
     free(secr_rate);
  if (chemotaxis)
     free(chemotaxis);
  if (haptokinesis)
     free(haptokinesis);
```

```
  if (min_conc)
     free(min_conc);
  if (max_conc)
     free(max_conc);
  if (cont_color)
     free(cont_color);
  if (grad_color)
     free(grad_color);
  (...)
}

void Parameter::Read(const char *filename) {
  (...)
  secr_rate = dgetparlist(fp, "secr_rate", n_chem, true);
  chemotaxis = dgetparlist(fp, "chemotaxis", n_chem, true);
  haptokinesis = dgetparlist(fp, "haptokinesis", n_chem, true);
  max_conc = dgetparlist(fp, "max_conc", n_chem, true);
  min_conc = dgetparlist(fp, "min_conc", n_chem, true);
  cont_color = dgetparlist(fp, "cont_color", n_chem, true);
  grad_color = dgetparlist(fp, "grad_color", n_chem, true);
  (...)
}

void Parameter::Write(ostream &os) const {
  (...)
  os << " secr_rate = "<< secr_rate[0] << endl;
  os << " chemotaxis = " << chemotaxis[0] << endl;
  os << " haptokinesis = " << haptokinesis[0] << endl;
  os << " max_conc = " << max_conc[0] << endl;
  os << " min_conc = " << min_conc[0] << endl;
  os << " cont_color = " << cont_color[0] << endl;
  os << " grad_color = " << grad_color[0] << endl;
  (...)
}
```

51. Change in `Parameter.h`:

```
class Parameter {

 public:
  double * secr_rate;
  double * chemotaxis;
  double * haptokinesis;
  double * max_conc;
  double * min_conc;
```

```
  double * cont_color;
  double * grad_color;
};
```

52. Add a new pde layer containing the ECM. In `ecm.cpp` add the following:

```
// Chemicals in ecm application
const int VEGF = 0;
const int ECM = 1;

// The layers in ecm application
int L_VEGF = -1;
int L_ECM = -1;
```

53. In the `INIT` block in `ecm.cpp`, add:

```
INIT {
    (...)

    // Assign layer indices to specific layer variables
    for (int l=0;l<PDEfield->Layers();l++){
      switch ((int) par.chem_type[l]) {
        case VEGF: L_VEGF=l;break;
        case ECM: L_ECM=l;break;
      }
    }
    (...)
}
```

54. Define the initial concentrations of the ECM field in the dish. There will be a uniform concentration of `init_conc` outside the vessel, an intermediate concentration inside the vessel, and a half-circular gradient around the gap outside the vessel as transition between the concentrations inside and outside the vessel Figure 2C).

```
INIT {

  try {

    (...)

    // Initialisation of PDE fields
    for (int x=0;x<par.sizex;x++)
    for (int y=0;y<par.sizey;y++)
```

```
      for (int l=0;l<PDEfield->Layers();l++){
        if (l==L_VEGF) {
          (...)

        /* ======================================================
        /  UNIFORM ECM DENSITY
        /  Uniform ECM density outside vessel
        /  Intermediate density inside vessel
        /  Behind gap circular gradient as transition to outside
        ======================================================*/
        } else if (l==L_ECM) {
          // If there is no wall uniform density
          if (!par.gapy)
            PDEfield->setValue(l,x,y,par.init_conc[l]);

          // If there is a wall and (x,y) lies behind it,
          // set to intermediate concentration
          else {
            double con = (par.max_conc[l]-par.min_conc[l])/2.;
            if (y > par.gapy)
              PDEfield->setValue(l,x,y,con);
            // If there is a wall and (x,y) lies outside,
            // set to outside density or (when near gap)
            // 'in between' density
            else {
              double dist2 = (x-par.sizex/2)*(x-par.sizex/2) +
                (y-par.gapy)*(y-par.gapy);
              double dist2max = (par.gapx)*(par.gapx)/4;
              // If (x,y) is outside half circle around gap
              // set to uniform outside density
              if (dist2 > dist2max)
                PDEfield->setValue(l,x,y,par.init_conc[l]);
              else
                PDEfield->setValue(l,x,y,con +
                  (par.init_conc[l]-con)*dist2/dist2max);
            }
          }
        } else {
            (...)
        }
      }
(...)

}
```

55. Just as with the VEGF field, the ECM field does not diffuse, so we add it

to the `skiplist` for the Diffuse function. For now we keep the `Secrete` function out-commented. In the `TIMESTEP` block in `ecm.cpp`, change the following code:

```
TIMESTEP {
  (...)
    static int SkipList[2] = {L_VEGF,L_ECM};
    if (i>=par.relaxation) {
      for (int r=0;r<par.pde_its;r++) {
        //dish->PDEfield->Secrete(dish->CPM);
        dish->PDEfield->Diffuse(1, SkipList, sizeof(SkipList));
      }
    }
  (...)
}
```

We will now add haptokinesis to the model, the phenomenon that cell motility depends on the concentration of ECM, with highest motility at intermediate ECM densities. We define the haptokinesis energy term with:

$$\Delta H_{\text{haptokinesis}} = \eta\delta(\vec{x},0)\left(-1+\frac{1}{\rho\sqrt{2pi}}e^{\frac{-(c_E(\vec{x}\prime)-\mu)^2}{2\rho^2}}\right), \tag{6}$$

where η is the haptokinesis strength anf $\mu = \frac{1}{2}$ is the intermediate ECM density; ECM densities in the model are in the range $c_E \in [0,1]$. The standard deviation ρ is set to a value $\rho = 0.2$ to ensure diversity of cell motility over the range of available ECM values.

56. In `ca.cpp` add the new Haptokinesis function (**see Note 9**):

```
int CellularPotts::DeltaH(int x,int y, int xp, int yp,
                          PDE *PDEfield)
{
   (...)
}

// Haptokinesis
// Cells have optimal motility at intermediate ECM densities
int CellularPotts::Haptokinesis(int x, int y, int xp, int yp,
                                PDE *PDEfield)
{
  double s = 0.2;
  double m = 0.5;
  double c = 0.0;
  int tmpDH = 0;
```

```
  for (int l=0;l<PDEfield->Layers();l++){
    if (par.haptokinesis[l]) {
      //Only during extension, look at concentration at cell site
      if (!sigma[x][y]) {
        c = PDEfield->Sigma(l,xp,yp);
        double c_scaled = (c-par.min_conc[l])/(par.max_conc[l] -
                          par.min_conc[l]);
        tmpDH += par.haptokinesis[l] *
                 (-1+exp(-1*(c_scaled-m)*(c_scaled-m)/(2*s*s)))/
                 sqrt(2*M_PI*s*s));
      }
    }
  }
  return tmpDH;
}
```

57. In `ca.h` add the (private) `Haptokinesis` function:

```
class CellularPotts {
  (...)
private:
  (...)
  int DeltaH(int x,int y, int xp, int yp, PDE *PDEfield=0);
  int Haptokinesis(int x, int y, int xp, int yp, PDE * PDEfield);
  (...)
};
```

58. In the `DeltaH` function in `ca.cpp`, we change the chemotaxis code to allow chemotaxis per pde layer. Just below the chemotaxis definition, we call the `Haptokinesis` function.

```
int CellularPotts::DeltaH(int x,int y, int xp, int yp,
                          PDE *PDEfield)
{
  (...)
  /* Chemotaxis */
  if (PDEfield && (par.vecadherinknockout ||
     (sxyp==0 || sxy==0))) {

    // copying from (xp, yp) into (x,y)
    // If par.extensiononly == true, apply CompuCell's method,
    // i.e. only chemotactic extensions contribute to
    // energy change
    if (!( par.extensiononly && sxyp==0)) {
```

```
      // Calculate chemotaxis and saturation for each layer
      for (int l=0;l<PDEfield->Layers();l++){
        int DDH = (int)(par.chemotaxis[l]*
                  (sat(PDEfield->Sigma(l,x,y)) -
                   sat(PDEfield->Sigma(l,xp,yp))));
        DH-=DDH;
      }
    }
  }

  // Haptokinesis
  // Cells have optimal motility at intermediate ECM densities
  if (PDEfield)
    DH -= Haptokinesis(x,y,xp,yp,PDEfield);
  (...)
}
```

59. Change the visualization code in block `TIMESTEP` in `ecm.cpp` in order to choose which layers to show. We allow only one layer for which we show the gradient (**See Note 10**).

```
TIMESTEP {
  (...)

    if (par.graphics && !(i%par.storage_stride)) {

      int tx,ty;

      BeginScene();

      // Plot gradients
      bool field_plotted = false;
      for (int l = 0; l<dish->PDEfield->Layers();l++)
        if (par.grad_color[l]) {
          dish->PDEfield->Plot(this,l);
          field_plotted = true;
          break;
        }
      // You need to call "ClearImage" if no PDE field
      // is plotted, because the CPM medium is considered
      // transparant
      if (!field_plotted) {
        ClearImage();
        dish->Plot(this);
      }
```

```
      // plot contours
      if (i>=par.relaxation)
        for (int l=0; l < dish->PDEfield->Layers();l++)
          if (par.cont_color[l])
            dish->PDEfield->
                  ContourPlot(this,l,par.cont_color[l]);
      dish->Plot(this);

      char title[400];
      (...)
    }


    if (par.store && !(i%par.storage_stride)) {
      (...)
      BeginScene();

      // Plot gradients
      bool field_plotted = false;
      for (int l = 0; l<dish->PDEfield->Layers();l++)
        if (par.grad_color[l]) {
          dish->PDEfield->Plot(this,l);
          field_plotted = true;
          break;
        }
      // You need to call "ClearImage" if no PDE field
      // is plotted, because the CPM medium is considered
      // transparant
      if (!field_plotted) {
        ClearImage();
        dish->Plot(this);
      }
      // plot contours
      if (i>=par.relaxation)
        for (int l=0; l < dish->PDEfield->Layers();l++)
          if (par.cont_color[l])
            dish->PDEfield->
                  ContourPlot(this,l,par.cont_color[l]);
      dish->Plot(this);

      EndScene();
      (...)
    }
  (...)
}
```

60. In the parameter file `ecm.par` we add and change the following parameters (**see Note 11**):

```
# PDE parameters
n_chem = 2
chem_type = 0,1
init_conc = 0.87,0.9
diff_coeff = 6e-11,0
decay_rate = 1e-3,0
secr_rate = 0,0
chemotaxis = 3500,0
haptokinesis = 0,100
min_conc = 0,0
max_conc = 2,1
cont_color = 0,0
grad_color = 0,1
```

61. Compile and run the code. You will now see that the cells will not migrate far out of the vessel, because the ECM concentrations are too high. Change the `haptokinesis` parameter to other values to see its effect.

3.4.4. Introducing interaction between layers: adding proteolysis to the model

In this section we add proteolysis: the degradation of ECM by matrix metalloproteinases (MMPs). The MMPs are secreted by ECs and the secretion depends on the concentration of VEGF. Therefore we have to change the parameter `secr_rate`, which is an array, to a table where we can define the secretion rate of one chemical based on the concentration of another chemical. Furthermore, the decay of the ECM depends on the concentration of MMPs. So we will also replace `decay_rate` with a table.

By introducing such a general way of handling interactions between layers, it is relatively easy to add or remove layers of chemicals without the need to create or remove parameters. For example, we could add a new rule to our model that cells also secrete VEGF themselves, and we only need to change an entry in the secretion table. The calculation of the concentrations in each layer can then be implemented in a similar general way, however to speed up computation time, we choose to restrict the calculation and only use those parameters we know we are going to use in our current model.

62. First, create the secretion and decay tables in the `data` directory (or to the directory `src` if you have copied all data files to `src`). We read these tables as follows: an entry > 0 in row i and column j means: the decay/secretion rate of chemical i is affected by the concentration of chemical j. The entries on the diagonal are the 'normal' decay and secretion rates. The row and

column indices stand for the indices we assign to the chemicals (VEGF=0, ECM=1 and MMP=2).

In `Decay.dat`, we define that the decay rate of VEGF and MMP is 1e-3 and the decay rate of ECM by MMP is 3e-3:

```
3
1e-3 0 0
0 0 3e-3
0 0 1e-3
```

In `Secr.dat` we set the secretion rate of MMPs induced by VEGF to 8e-5:

```
3
0 0 0
0 0 0
8e-5 0 0
```

63. Introduce two new parameters, `DTable` and `STable` which will store the secretion and decay tables' filename. Add to `parameter.cpp`:

```
Parameter::Parameter() {
  (...)
  Jtable = strdup("J.dat");
  Dtable = strdup("notable");
  Stable = strdup("notable");
  (...)
}

void Parameter::CleanUp(void) {
  if (Jtable)
     free(Jtable);
  if (Dtable)
     free(Dtable);
  if (Stable)
     free(Stable);
  (...)
}

void Parameter::Read(const char *filename) {
  (...)
  Jtable = sgetpar(fp, "Jtable", "J.dat", true);
  Dtable = sgetpar(fp, "Dtable", "notable", true);
  Stable = sgetpar(fp, "Stable", "notable", true);
```

```
  (...)
}

void Parameter::Write(ostream &os) const {
  (...)

  if (Jtable)
    os << " Jtable = " << Jtable << endl;
  if (Dtable)
    os << " Dtable = " << Dtable << endl;
  if (Stable)
    os << " Stable = " << Stable << endl;
  (...)
}
```

64. In `parameter.h` add:

```
class Parameter {

 public:
  (...)
  char * Jtable;
  char * Dtable;
  char * Stable;
  (...)
};
```

65. In `parameter.cpp` and `parameter.h`, remove all lines related to the decay_rate and secr_rate parameters.
66. Add the function `ReadStaticTable` to `pde.cpp`. This function reads in the secretion and decay tables and translates them to tables for all layers that are used in the model:

```
void PDE::Diffuse(int repeat, const int * skiplist, int nrskips) {(...)}

// Create table to store secretion or decay rates
double ** PDE::ReadStaticTable(const char *fname, double ** tbl) {

  int n, m; // n=number of chem layers, m=number of types in file
  n = par.n_chem;

  // Allocate memory
  if (tbl) { free(tbl[0]); free(tbl); }
  tbl=(double **)malloc(n*sizeof(double *));
```

```
tbl[0]=(double *)malloc(n*(n+1)*sizeof(double));
for (int i=1;i<n;i++) tbl[i]=tbl[i-1]+(n+1);

// If no file with table is found,
// fill in an empty table
ifstream jtab(fname);
if (!jtab){
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      tbl[i][j+1]=0.0;

} else {

  double **tmptbl=0;
  jtab >> m;

  // Allocate memory for temp chemtype table
  tmptbl=(double **)malloc(m*sizeof(double *));
  tmptbl[0]=(double *)malloc(m*m*sizeof(double));
  for (int i=1;i<m;i++) {
    tmptbl[i]=tmptbl[i-1]+m;
  }
  for (int i=0;i<m;i++)
    for (int j=0;j<m;j++)
      jtab >> tmptbl[i][j];

  // Fill chemlayer table with values
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++){
      int k,l;
      k = par.chem_type[i];
      l = par.chem_type[j];
      if (k<m && l<m) tbl[i][j+1]=tmptbl[k][l];
      else tbl[i][j+1]=0.0;
    }

  // First column indicates if whole row is 0 or not
  // This speeds up the simulation
  for (int i=0;i<n;i++){
    double toti = 0.0;
    for (int j=0;j<n;j++) toti+=tbl[i][j+1];
    if (toti==0.0) tbl[i][0]=0;
    else tbl[i][0]=1;
  }

  free(tmptbl[0]);
```

```
    free(tmptbl);
  }
  return tbl;
}
```

67. Include `fstream` at the beginning of `pde.cpp`:

```
#include <cstdlib>
#include <fstream>
```

68. Add the variables `SecrTable` and `DegrTable`. They are read from the table files during initialization of the pde object. Add the following lines of code in `pde.cpp`:

```
PDE::PDE(const int l, const int sx, const int sy) {

  sigma=0;
  thetime=0;

  DegrTable=0;
  SecrTable=0;
  DegrTable=ReadStaticTable(par.Dtable, DegrTable);
  SecrTable=ReadStaticTable(par.Stable, SecrTable);

  (...)
}
```

69. Add the following lines to the destructor:

```
// destructor (virtual)
PDE::~PDE(void) {

  if (DegrTable){
    free(DegrTable[0]);
    free(DegrTable);
    DegrTable = 0;
  }

  if (SecrTable){
    free(SecrTable[0]);
    free(SecrTable);
    SecrTable = 0;
  }

  (...)
}
```

70. To get the secretion and decay rates, call the functions `GetSecrTable` and `GetDegrTable`. They are defined as inline functions in `pde.h`:

```
class PDE {
  (...)
 public:
  (...)

  void Secrete(CellularPotts *cpm);

  //! Get the secretion rate for layer i induced by layer j
  inline double GetSecrTable(int i, int j){
    return SecrTable[i][j+1];}

  //! Get the decay rate for layer i induced by layer j
  inline double GetDegrTable(int i, int j){
    return DegrTable[i][j+1];}

  (...)
};
```

71. Add also to `pde.h` the other new functions:

```
class PDE {
  (...)
 protected:
  (...)
  virtual double ***AllocateSigma(const int layers, const int sx,
                                  const int sy);
  //! table with decay rates
  double ** DegrTable;

  //! table with secretion rates
  double ** SecrTable;

  /*! \brief Read a table of static factors (JD).

  \param fname: Filename with table.
  First line: number of chem_types (including medium)
  Next lines: matrix with elements Fij where Fij>0 means:
  chem in layer j degrades/induces the secretion of chem
  in layer   i.
  \param tbl: The (n_chem x n_chem+1) matrix that will
  be returned with values from table
  */
```

```
  double **  ReadStaticTable(const char *fname, double ** tbl);
  (...)
};
```

72. Add the MMP chemical and its layer. In `ecm.cpp`, add the following lines of code:

```
// Chemicals in ecm application
const int VEGF = 0;
const int ECM = 1;
const int MMP = 2;

// The layers in ecm application
int L_VEGF = -1;
int L_ECM = -1;
int L_MMP = -1;
```

73. In the `INIT` block in `ecm.cpp` add:

```
INIT {
    (...)

 // Assign layer indices to specific layer variables
    for (int l=0;l<PDEfield->Layers();l++){
      switch ((int) par.chem_type[l]) {
        case VEGF: L_VEGF=l;break;
        case ECM: L_ECM=l;break;
        case MMP: L_MMP=l;break;
      }
    }
    (...)
}
```

MMPs degrade the ECM at a rate proportional to the ECM concentrations. The PDE that defines the evolution of the ECM field is given by:

$$\frac{\partial c_E(\vec{x},t)}{\partial t} = -\delta(\sigma(\vec{x}),0)\epsilon_{EM} c_M(\vec{x},t) c_E(\vec{x},t), \tag{7}$$

where $c_E(\vec{x},t)$ and $c_M(\vec{x},t)$ represent the concentrations of ECM and MMPs, and ϵ_{EM} is a degradation constant.

ECs secrete MMPs at a rate proportional to the local VEGF concentrations. The PDE describing the changes in the MMP concentrations is given by:

$$\frac{\partial c_M(\vec{x},t)}{\partial t} = \alpha_{MV} c_V(\vec{x},t)\left(1-\delta(\sigma(\vec{x}),0)\right) H\left(c_{M,\max}-c_{M(\vec{x},t)}\right) - \delta(\sigma(\vec{x}),0)\epsilon_M c_M(\vec{x},t) + D_M \nabla^2 c_M(\vec{x},t), \quad (8)$$

where $c_V(\vec{x},t)$ represents the concentration of VEGF, and α_{MV} is the secretion rate of MMPs, ϵ_M is the decay rate of MMPs, and D_M is the diffusion coefficient of MMPs. The Kronecker-deltas state that cells only secrete MMPs at the lattice sites they cover, while the ECM is only degraded outside the cells. The Heaviside step function, $H\left(c_{M,\max}-c_M(\vec{x},t)\right)$, suppresses secretion of MMPs if the local MMP concentration exceeds the maximum concentration of $c_{M,\max}=1$.

74. Implement these PDE's by redefining the `Secrete` function in `ecm.cpp`:

```
void PDE::Secrete(CellularPotts *cpm) {

  const double dt=par.dt;
  double ***tmp;

  // Check if ECM, MMP and VEGF layers are present
  if (L_ECM==-1||L_MMP==-1||L_VEGF==-1){
    cerr << "ECM, MMP and/or VEGF layer is missing ";
    cerr << "in parameter file\n";
    exit(1);
  }

  // Take those secr and decay rates from the table that are
  // actually used in the simulation
  static const double d_ECM_MMP = GetDegrTable(L_ECM,L_MMP);
  static const double s_MMP_VEGF = GetSecrTable(L_MMP,L_VEGF);
  static const double d_MMP = GetDegrTable(L_MMP,L_MMP);

  for (int x=0;x<sizex;x++)
    for (int y=0;y<sizey;y++){
      if (cpm->Sigma(x,y)>0){
        // Inside cells: secretion
        // Decay is not allowed at cell sites
          alt_sigma[L_ECM][x][y] = sigma[L_ECM][x][y] ;
          alt_sigma[L_MMP][x][y] = sigma[L_MMP][x][y] +
            s_MMP_VEGF*sigma[L_VEGF][x][y]*dt;
          alt_sigma[L_VEGF][x][y] = sigma[L_VEGF][x][y];
      }
      // Outside cells: decay
```

```
      else {
        alt_sigma[L_ECM][x][y] = sigma[L_ECM][x][y]
          - sigma[L_ECM][x][y]*dt*d_ECM_MMP*sigma[L_MMP][x][y];
        alt_sigma[L_MMP][x][y] = sigma[L_MMP][x][y]
          - sigma[L_MMP][x][y]*dt*d_MMP;
        alt_sigma[L_VEGF][x][y] = sigma[L_VEGF][x][y];
      }

      // Check if concentrations exceed min and max values
      // and correct them if necessary
      for (int l=0;l<layers;l++){
        if (alt_sigma[l][x][y]>par.max_conc[l])
          alt_sigma[l][x][y] = par.max_conc[l];
        else if (alt_sigma[l][x][y]<par.min_conc[l])
          alt_sigma[l][x][y] = par.min_conc[l];
      }
    }

  tmp=sigma;
  sigma=alt_sigma;
  alt_sigma=tmp;
}
```

75. Call the new `Secrete` function in the `TIMESTEP` block in `ecm.cpp`:

```
TIMESTEP {
  (...)
    static int SkipList[2] = {L_VEGF,L_ECM};
    if (i>=par.relaxation) {
      for (int r=0;r<par.pde_its;r++) {
        dish->PDEfield->Secrete(dish->CPM);
        dish->PDEfield->Diffuse(1, SkipList, sizeof(SkipList));
      }
    }
  (...)
}
```

76. Change the initialization of the VEGF field in `ecm.cpp` in the `INIT` block:

```
INIT {
  (...)

  if (l==L_VEGF) {

          double D = par.diff_coeff[l];
```

```
          double d = PDEfield->GetDegrTable(l,l);
  (...)
}
```

77. In the parameter file `ecm.par` add the new parameters and remove the `decay_rate` and `secr_rate` parameters. Also add the MMP layer:

```
Jtable = J.dat
Stable = Secr.dat
Dtable = Decay.dat
(...)

# PDE parameters
n_chem = 3
chem_type = 0,1,2
init_conc = 0.87,0.9,0
diff_coeff = 6e-11,0,1e-14
chemotaxis = 3500,0,0
haptokinesis = 0,100,0
min_conc = 0,0,0
max_conc = 1,1,1
cont_color = 0,0,0
grad_color = 0,1,0
```

78. Compile and run the code. You will now see that the ECM layer is degraded by the MMPs that cells secrete. The cells will show some branching behavior. Some cells split off and migrate to the tumor, but the sprout itself will not be able to grow this far. You can change the ECM_MMP decay rate to see its effect.

3.4.5. Adding proliferation to create a growing sprout

By adding proteolysis we were able create a growing and branching sprout, but it was unable to reach the tumor. For this we add proliferation to the model. We let cells divide, when their proportion of cell surface NOT touching other cells is larger than a certain threshold.

79. First add three new parameters: `allowdiv`, which allows proliferation if set to true, `growthrate`, which determines the rate with which the cells grow and `ECMsurfratio`, which is minimum proportion of the cell surface connecting with the ECM to allow division. Add the following lines to `parameter.cpp`:

```
Parameter::Parameter() {
  (...)
```

```
  border_energy = 100;
  allowdiv = false;
  growthrate = 2;
  ECMsurfratio = 0.6;
  (...)
}

void Parameter::Read(const char *filename) {
  (...)
  border_energy = igetpar(fp, "border_energy", 100, true);
  allowdiv = bgetpar(fp, "allowdiv", false, true);
  growthrate = igetpar(fp, "growthrate", 2, true);
  ECMsurfratio = fgetpar(fp, "ECMsurfratio", 0.6, true);
  (...)
}

void Parameter::Write(ostream &os) const {
  (...)
  os << " border_energy = " << border_energy << endl;
  os << " allowdiv = " << allowdiv << endl;
  os << " growthrate = " << growthrate << endl;
  os << " ECMsurfratio = " << ECMsurfratio << endl;
  (...)
}
```

80. Add to `parameter.h`:

```
class Parameter {

 public:
  (...)
  int border_energy;
  bool allowdiv;
  int growthrate;
  double ECMsurfratio;
  (...)
};
```

81. Create the new functions `GrowAndDivideECs` and `MeasureCellSurfaces` in `ca.cpp`:

```
void CellularPotts::GrowAndDivideCells(int growth_rate) {(...)}


// Grow cells and divide those cells (with certain probability)
```

```
// when ratio (cell-ECM surface)/(total surface) is above threshold
void CellularPotts::GrowAndDivideECs(int growth_rate,
                                     double ECMsurfratio) {

  //calculate length growth rate
  double growth_rate_length;
  growth_rate_length = ((double)par.target_length/2)/
                       (((double) par.target_area/2)/growth_rate);
  MeasureCellSurfaces();

  vector<Cell>::iterator c=cell->begin(); ++c;
  vector<bool> which_cells(cell->size());

  for (;c!=cell->end(); c++) {
    c->SetTargetArea(min(par.target_area,c->
         TargetArea()+growth_rate));
    c->SetTargetLength(min((double)par.target_length,
         c->TargetLength()+growth_rate_length));
    double r = (double)c->ECMSurface()/c->TotSurface();
    if ((c->Area()>=par.target_area) && r > ECMsurfratio) {
      if (RANDOM()< r-ECMsurfratio)
        which_cells[c->Sigma()]=true;
      else
        which_cells[c->Sigma()]=false;
    } else {
      which_cells[c->Sigma()]=false;
    }
  }
  DivideCells(which_cells);
}

// Measure per cell the ratio (cell-ECM surface)/total surface
// When this value is above par.ECMsurfratio, the cell can divide
void CellularPotts::MeasureCellSurfaces(void){
  // Clean ECMsurfaces of all cells, including medium
  for (vector<Cell>::iterator c=cell->begin();c!=cell->end();c++) {
    c->SetECMSurface(0);
    c->SetTotSurface(0);
  }

  for (int x=1;x<sizex-1;x++)
  for (int y=1;y<sizey-1;y++) {
    if (sigma[x][y]>0) {
        for (int i=1;i<=4;i++) {
          int x2,y2;
          x2=x+nx[i]; y2=y+ny[i];
```

```
          if ((!sigma[x2][y2]) &&
              (par.gapy==0 || y2 < par.gapy-par.target_length)){
            (*cell)[sigma[x][y]].IncrementTotSurface();
            (*cell)[sigma[x][y]].IncrementECMSurface();
          }
          else if ((sigma[x2][y2]!=sigma[x][y]))
            (*cell)[sigma[x][y]].IncrementTotSurface();
        }
    }
  }
}
```

Add the functions to `ca.h`:

```
class CellularPotts {
  (...)
public:
  (...)
  void GrowAndDivideCells(int growth_rate);
  void GrowAndDivideECs(int growth_rate, double ECMsurfratio);
  void MeasureCellSurfaces(void);
  (...)
};
```

82. In `cell.h` add new functions and variables for calculating the total cell surface and the cell surface touching the ECM:

```
class Cell
{
(...)

public:
  (...)
  inline void SetECMSurface(int s){ecm_surf = s;}
  inline int ECMSurface() {return ecm_surf;}
  inline void IncrementECMSurface(){ecm_surf++;};
  inline void SetTotSurface(int s){tot_surf = s;}
  inline int TotSurface() {return tot_surf;}
  inline void IncrementTotSurface(){tot_surf++;};
  (...)

protected:

  int ecm_surf; //cell surface connected to ECM
  int tot_surf; //total cell surface
```

83. Add a call to the `GrowAndDivideECs` function to `ecm.cpp` in the `TIMESTEP` block:

```
TIMESTEP {
  (...)

    dish->CPM->AmoebaeMove(dish->PDEfield);

    // Grow cells and divide those with a ratio
    // (cell-ECM surface)/(total surface) above threshold
    // ECMsurfratio
    if (par.allowdiv && i>=par.relaxation) {
      if (!(i%5)) {
        dish->CPM->GrowAndDivideECs(par.growthrate,
                    par.ECMsurfratio);
      }
    }
  (...)
}
```

84. In the parameter file, set the following values:

```
extensiononly = true

# cell growth and division
allowdiv = true
growthrate = 2
ECMsurfratio = 0.73
```

3.4.6. One more ECM interaction: adding haptotaxis to the model

The last ingredient we are going to add to the model is *haptotaxis*, the movement of cells up to higher ECM gradients. The haptotaxis energy term is given by:

$$\Delta H_{\text{haptotaxis}} = -\Gamma\delta(\sigma(\vec{x}),0)\left(\frac{c_E(\vec{x})}{1+sc_E(\vec{x})} - \frac{c_E(\vec{x\prime})}{1+sc_E(\vec{x\prime})}\right), \tag{9}$$

with $c_E(\vec{x})$, the local ECM density, Γ, the strength of the haptotactic response, and s, a saturation parameter. The saturation term reduces haptotaxis at high ECM concentrations.

In the implementation, haptotaxis is similar to chemotaxis and we can use the existing chemotaxis code for it. However, we want to set different saturation factors for chemotaxis up the VEGF gradient (`saturation = 0`) and haptotaxis up the ECM gradient (`saturation = 7`).

85. First change the `saturation` parameter to allow values for multiple layers. In `parameter.cpp` change and add the following code:

```
Parameter::Parameter() {
  (...)
  saturation = new double[1];
  saturation[0] = 0.;
  (...)
}

void Parameter::CleanUp(void) {
  (...)
  if (grad_color)
     free(grad_color);
  if (saturation)
     free(saturation);
}

void Parameter::Read(const char *filename) {
  (...)
  saturation = dgetparlist(fp, "saturation", n_chem, true);
  (...)
}

void Parameter::Write(ostream &os) const {
  (...)
  os << " saturation = " << saturation[0] << endl;
  (...)
}
```

86. Change in `parameter.h` the following line:

```
class Parameter {

 public:
  (...)
  double  * saturation;
  (...)
}
```

87. Add a `Saturation` function to `ca.cpp` to allow saturation per layer and change the chemotaxis code to call this new function:

```
int CellularPotts::DeltaH(int x,int y, int xp, int yp, PDE *PDEfield)
```

```
{
  (...)
  /* Chemotaxis */
  if (PDEfield && (par.vecadherinknockout ||
                   (sxyp==0 || sxy==0))) {

    if (!( par.extensiononly && sxyp==0)) {
      // Calculate chemotaxis and saturation for each layer
      for (int l=0;l<PDEfield->Layers();l++){
        int DDH = (int)(par.chemotaxis[l]*
                    (sat(PDEfield->Sigma(l,x,y),l)
                  - sat(PDEfield->Sigma(l,xp,yp),l)));
        DH-=DDH;
      }
    }
  }
}
```

88. Finally, change the following parameters in `ecm.par`:

```
chemotaxis = 3500,300,0
saturation = 0,0,7
```

89. Compile and run the complete model.

4. Notes

Note 1: The Windows compilation has been tested on Windows 7. Lisanne Rens is thanked for performing the test and improving the instructions.

Note 2: For some parameters (*e.g.*, high diffusion coefficients) the diffusion algorithm will become numerically unstable; in those cases you can experiment with the number and size of PDE steps carried out after each Monte Carlo Step, using parameters dt and pde_its.

Note 3: An interesting experiment is to run the simulation in the absence of contact inhibition for these parameter values, by setting `vecadherinknockout = true`. A network will form temporarily, but it will collapse eventually.

Note 4: Color definitions for cells and contour lines are in the text file 'default.ctb'. The format is '[index] [red] [green] [blue]', where index is an integer corresponding with the cell type, , and the '[red] [green] [blue]' define the RGB channels in the range [0-255]. Put 'default.ctb' in the same folder as the executable.

Note 5: Parts of the existing code are shown to give an indication of the placement of the new code. `(...)` marks a part of the code that is not shown.

Note 6: In your simulation you will now see the VEGF gradient colored in grey and green contour lines. In Figure 3A and 3B of [19] the background is white, showing the ECM concentrations, which are set to zero initially. Later on, we will add functionality to color a layer of choice.

Note 7: Make sure to place all these parameters under `n_chem`!

Note 8: Here we change the chemotaxis parameter, so make sure that the old definition (single integer value) is removed!

Note 9: The original code that produced the simulations in Daub and Merks [19] contained an error, resulting in favoring cell protrusions into the ECM over all other possible copy attempts (e.g., retractions). With different chemotaxis and haptokinesis parameters here than those used in [19] similar results are still obtained by using the corrected code. The following code was used in [19]:

```
// Haptokinesis
// Cells have optimal motility at intermediate ECM densities
int CellularPotts::Haptokinesis(int x, int y, int xp, int yp,
                                PDE * PDEfield)
{
  double s = 0.2;
  double m = 0.5;
  double c = 0.0;
  int tmpDH = 0;

  for (int l=0;l<PDEfield->Layers();l++){
    if (par.haptokinesis[l]) {
      //Only during extension, look at concentration at cell site
      if (!sigma[x][y])
        //if no extension, set c to min_conc to have max penalty
        c = par.min_conc[l];
      else
        c = PDEfield->Sigma(l,xp,yp);

      double c_scaled = (c-par.min_conc[l]) / (par.max_conc[l] -
                        par.min_conc[l]);
      tmpDH += par.haptokinesis[l] *
               (-1+exp(-1*(c_scaled-m)*(c_scaled-m)/(2*s*s))/
               sqrt(2*M_PI*s*s));
    }
  }
  return tmpDH;
}
```

Note 10: In the current implementation you cannot specifically set the gradient color, you indicate with a number > 0 in the parameter `grad_color`, which layer will be shown as gradient.

Note 11: We moved the chemotaxis parameter to the PDE parameters part, to keep all parameters with values per layer together.

5. References

1. Kitano H (2002) Systems Biology: A Brief Overview. Science (New York, NY) 295: 1662–1664.

2. Folkman J, Hauenschild C (1980) Angiogenesis in vitro. Nature 288: 551–556.

3. Califano J, Reinhart-King C (2008) A balance of substrate mechanics and matrix chemistry regulates endothelial cell network assembly. Cell Mol Bioeng 1: 122–132. doi:10.1007/s12195-008-0022-x.

4. Oster GF, Murray JD, Harris AK (1983) Mechanical aspects of mesenchymal morphogenesis. J Embryol Exp Morphol 78: 83–125.

5. Manoussaki D, Lubkin S, Vernon R, Murray J (1996) A mechanical model for the formation of vascular networks in vitro. Acta Biotheor 44: 271–282.

6. Manoussaki D (2003) A mechanochemical model of angiogenesis and vasculogenesis. ESAIM: Math Model Num 37: 581–599. doi:10.1051/m2an:2003046.

7. Gamba A, Ambrosi D, Coniglio A, de Candia A, Di Talia S, et al. (2003) Percolation, morphogenesis, and Burgers dynamics in blood vessels formation. Phys Rev Lett 90: 118101. doi:10.1103/PhysRevLett.90.118101.

8. Serini G, Ambrosi D, Giraudo E, Gamba A, Preziosi L, et al. (2003) Modeling the early stages of vascular network assembly. EMBO J 22: 1771–1779. doi:10.1093/emboj/cdg176.

9. Ambrosi D, Gamba A, Serini G (2004) Cell directional and chemotaxis in vascular morphogenesis. B Math Biol 66: 1851–1873. doi:10.1016/j.blum.2004.04.004.

10. Keller E (1970) Initiation of slime mold aggregation viewed as an instability. J Theor Biol 26: 399–415.

11. R M H Merks, J A Glazier (2005) A cell-centered approach to developmental biology. Physica A 352: 113–130. doi:10.1016/j.physa.2004.12.028.

12. Graner F, Glazier JA (1992) Simulation of biological cell sorting using a two-dimensional extended Potts model. Phys Rev Lett 69: 2013–2016.

13. Merks RMH, Brodsky SV, Goligorksy MS, Newman SA, Glazier JA (2006) Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling. Dev Biol 289: 44–54. doi:10.1016/j.ydbio.2005.10.003.

14. Palm MM, Merks RMH (2013) Vascular networks due to dynamically arrested crystalline ordering of elongated cells. Phys Rev E 87: 012725. doi:10.1103/PhysRevE.87.012725.

15. Merks RMH, Perryn ED, Shirinifard A, Glazier JA (2008) Contact-inhibited chemotaxis in de novo and sprouting blood-vessel growth. PLoS Comp Biol 4: e1000163. doi:10.1371/journal.pcbi.1000163.

16. Szabó A, Mehes E, Kosa E, Czirok A (2008) Multicellular sprouting in vitro. Biophys J 95: 2702–2710. doi:10.1529/biophysj.108.129668.

17. Köhn-Luque A, De Back W, Starruß J, Mattiotti A, Deutsch A, et al. (2011) Early embryonic vascular patterning by matrix-mediated paracrine signalling: A mathematical model study. PLoS ONE 6: e24175. doi:10.1371/journal.pone.0024175.t001.

18. Köhn-Luque A, de Back W, Yamaguchi Y, Yoshimura K, Herrero MA, et al. (2013) Dynamics of VEGF matrix-retention in vascular network patterning. Phys Biol 10: 066007. doi:10.1088/1478-3975/10/6/066007.

19. Daub JT, Merks RMH (2013) A cell-based model of extracellular-matrix-guided endothelial cell migration during angiogenesis. B Math Biol 75: 1377–1399. doi:10.1007/s11538-013-9826-5.

20. Scianna M, Munaron L, Preziosi L (2011) A multiscale hybrid approach for vasculogenesis and related potential blocking therapies. Prog Biophys Mol Bio 106: 450–462. doi:doi: 10.1016/j.pbiomolbio.2011.01.004.

21. Boas SEM, Merks RMH (2014) Synergy of cell-cell repulsion and vacuolation in a computational model of lumen formation. Journal of The Royal Society Interface 11: 20131049–20131049. doi:10.1038/ncb1705.

22. Shirinifard A, Gens JS, Zaitlen BL, Popławski NJ, Swat M, et al. (2009) 3D multi-cell simulation of tumor growth and angiogenesis. PLoS ONE 4: e7190. doi:10.1371/journal.pone.0007190.

23. Shirinifard A, Glazier JA, Swat M, Gens JS, Family F, et al. (2012) Adhesion Failures Determine the Pattern of Choroidal Neovascularization in the Eye: A Computer Simulation Study. PLoS Comput Biol 8: e1002440. doi:10.1371/journal.pcbi.1002440.s022.

24. Kleinstreuer N, Dix D, Rountree M, Baker N, Sipes N, et al. (2013) A computational model predicting disruption of blood vessel development. PLoS Comput Biol 9: e1002996. doi:10.1371/journal.pcbi.1002996.s011.

25. Bauer AL, Jackson TL, Jiang Y (2007) A cell-based model exhibiting branching and anastomosis during tumor-induced angiogenesis. Biophys J 92: 3105–3121. doi:10.1529/biophysj.106.101501.

26. Bauer AL, Jackson TL, Jiang Y (2009) Topography of extracellular matrix mediates vascular morphogenesis and migration speeds in angiogenesis. PLoS Comput Biol 5: e1000445. doi:10.1371/journal.pcbi.1000445.

27. Scianna M, Bell CG, Preziosi L (2013) A review of mathematical models for the formation of vascular networks. J Theor Biol 333: 174–209. doi:10.1016/j.jtbi.2013.04.037.

28. Czirok A (2013) Endothelial cell motility, coordination and pattern formation during vasculogenesis. Wiley Interdiscip Rev Syst Biol Med 5: 587–602. doi:10.1002/wsbm.1233.

29. Wacker A, Gerhardt H (2011) Endothelial development taking shape. Current Opinion in Cell Biology.

30. Swat MH, Thomas GL, Belmonte JM, Shirinifard A, Hmeljak D, et al. (2012) Multi-Scale Modeling of Tissues Using CompuCell3D. Elsevier Inc. 42 pp. doi:10.1016/B978-0-12-388403-9.00013-8.

31. Szabó A, Varga K, Garay T, Hegedűs B, Czirok A (2012) Invasion from a cell aggregate–the roles of active cell motion and mechanical equilibrium. Phys Biol 9: 016010–016010. doi:10.1088/1478-3975/9/1/016010.

32. van Oers RFM, Ruimerman R, Tanck E, Hilbers PAJ, Huiskes R (2008) A unified theory for osteonal and hemi-osteonal remodeling. Bone 42: 250–259. doi:10.1016/j.bone.2007.10.009.

33. Starruß J, De Back W, Brusch L, Deutsch A (2014) Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology. Bioinformatics 30:1331-1332. doi:10.1093/bioinformatics/btt772.

34. Pitt-Francis J, Pathmanathan P, Bernabeu MO, Bordas R, Cooper J, et al. (2009) Chaste: A test-driven approach to software development for biological modelling. Comput Phys Commun 180: 2452–2471. doi:10.1016/j.cpc.2009.07.019.

35. Roeland M H Merks, Guravage M, Inze D, Beemster GTS (2011) VirtualLeaf: An Open-Source Framework for Cell-Based Modeling of Plant Tissue Growth and Development. Plant Physiol 155: 656–666. doi:10.1104/pp.110.167619.

36. Holcombe M, Adra S, Bicak M, Chin S, Coakley S, et al. (2012) Modelling complex biological systems using an agent-based approach. Integr Biol 4: 53–64.

37. Glazier JA, Graner F (1993) Simulation of the differential adhesion driven rearrangement of biological cells. Phys Rev E 47: 2128–2154.

38. Savill NJ, Hogeweg P (1997) Modelling morphogenesis: From single cells to crawling slugs. J Theor Biol 184: 229–235.

39. Eden M (1961) A Two-Dimensional Growth Process. Proc Fourth Berkeley Symp on Math Statist and Prob 4: 223–239.

40. Merks RMH, Glazier JA (2006) Dynamic mechanisms of blood vessel growth. Nonlinearity 19: C1–C10. doi:10.1088/0951-7715/19/1/000.

41. Folkman J (2007) Angiogenesis: an organizing principle for drug discovery? Nat Rev Drug Discov 6: 273–286. doi:doi:10.1038/nrd2115.

42. Pepper MS (2001) Role of the matrix metalloproteinase and plasminogen activator-plasmin systems in angiogenesis. Arterioscler Thromb Vasc Biol 21: 1104–1117.

43. van Hinsbergh VWM, Koolwijk P (2008) Endothelial sprouting and angiogenesis: matrix metalloproteinases in the lead. Cardiovascular Research 78: 203–212. doi:10.1093/cvr/cvm102.

44. Gerhardt H, Betsholtz C (2003) Endothelial-pericyte interactions in angiogenesis. Cell Tissue Res 314: 15–23. doi:10.1007/s00441-003-0745-x.

45. Gerhardt H (2008) VEGF and endothelial guidance in angiogenic sprouting. Organogenesis 4: 241–246.

46. Senger DR, Perruzzi CA, Streit M, Koteliansky VE, de Fougerolles AR, et al. (2002) The $\alpha_1\beta_1$ and $\alpha_2\beta_1$ integrins provide critical support for vascular endothelial growth factor signaling, endothelial cell migration, and tumor angiogenesis. Am J Pathol 160: 195–204.

47. Lamalice L, Le Boeuf F, Huot J (2007) Endothelial cell migration during angiogenesis. Circulation research 100: 782–794. doi:10.1161/01.RES.0000259593.07661.1e.

48. DiMilla PA, Stone JA, Quinn JA, Albelda SM, Lauffenburger DA (1993) Maximal migration of human smooth muscle cells on fibronectin and type IV collagen occurs at an intermediate attachment strength. 122: 729–737.

49. Cox E, Sastry S, Huttenlocher A (2001) Integrin-mediated adhesion regulates cell polarity and membrane protrusion through the Rho family of GTPases. Mol Biol Cell 12: 265–277.

50. Coomber BL, Gotlieb AI (1990) In vitro endothelial wound repair. Interaction of cell migration and proliferation. Arteriosclerosis 10: 215–222.

51. Keller EF, Segel LA (1971) Model for chemotaxis. J Theor Biol 30: 225–234. doi:10.1016/0022-5193(71)90050-6.

Figure Captions

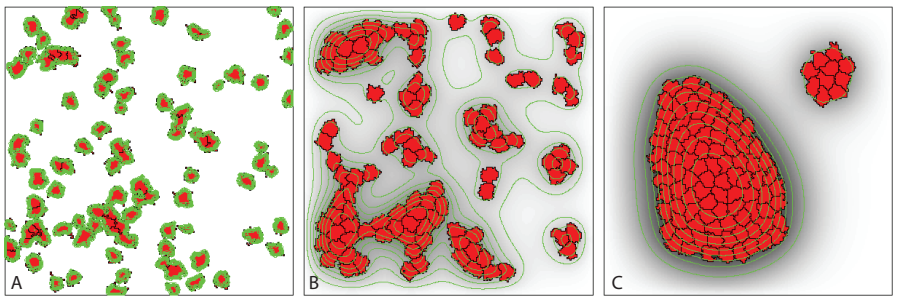

Figure 1: Typical output of TST-model `vessel` with parameter file `chemotaxis.par`. The model is a Cellular Potts implementation of the Keller-Segel model [51] of cellular aggregation due to chemotaxis. All cells secrete a chemoattractant, which diffuses and is degraded in the surrounding matrix. (A) Initial condition; (B) configuration after 500 Monte Carlo Steps; (C) configuration after 10000 Monte Carlo Steps. Grey levels indicate chemoattractant concentrations (white: low concentration; black: high concentration); contour lines connect equal chemoattractant concentrations; cells shown as red patches delineated with black contours.

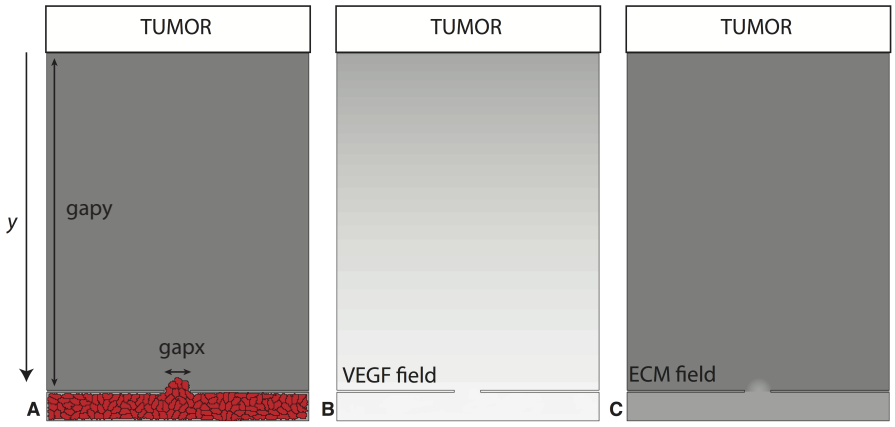

Figure 2: Setup of model domain. A Initial ECM field (grey) and model configuration: ECs are placed behind a vessel wall and can migrate through a gap in the wall. B Steady state VEGF field; C Initial ECM field. Adapted from Figure 1 in [19].

Table 1 Description of TST's standard parameter file

Name in parameter file	Symbol	Default value	Use
Parameters for standard Cellular Potts model			
`T`	T	`50`	Cellular temperature or cell motility parameter
`target_area`	A	`50`	Resting area of cells
`target_length`	L	`0`	Target length of cells
`lambda`	λ_A	`50`	Strength of area constraint
`lambda2`	λ_L	0	Strength of length constraint
`JTable`	-	`J.dat`	File containing values of J
`conn_diss`	H_0	`0`	Energy dissipation during update that would destroy connectivity
`vecadherinknockout`	-	`true`	True if the chemotaxis algorithm is NOT contact-inhibited (see [15])
`chemotaxis`	χ	`0`	Strength of chemotactic force (set to negative value for chemorepulsion)
`border_energy`	$J_{c,B}$	`100`	Contact energy between cells and borders. Default value creates non-sticky boundaries. Set to make cells adhere to boundaries.
`neighbours`	-	`3`	neighborhood order; warning: set to '2' to length constraint is used.
`periodic_boundaries`	-	`false`	If set to 'true' simulation uses periodic boundaries for CPM and PDE components
`extensiononly`	-	`false`	If set to 'true' simulation uses extension-only chemotaxis.
Parameters for partial-differential equation solver and CPM-PDE coupling			
`n_chem`	-	`1`	Number of fields in PDE component
`diff_coeff`	D	`1e-13`	Array of diffusion coefficients, typically in m^2s^{-1}. If more than one field is used, separate values with commas: e.g., 1e-13, 1e-11, 0
`decay_rate`	ϵ	`1.8e-4`	Array of degradation rates of chemicals, typically in s^{-1}. If more than one field is used, separate values with commas.

secr_rate	α	1.8e-4	Array of secretion rates of chemicals, typically in s^{-1} per lattice site. If more than one field is used, separate values with commas.
saturation	s	0	Saturation of chemotactic response
dt	Δt	2	Timestep of finite-difference integrator of PDE component. Typically in s.
dx	Δx	2.0e-6	Size of pixels for finite-difference integrator of PDE component. Typically in m.
pde_its	-	15	Number of finite-difference integration steps between two CPM Monte Carlo steps (MCS). I.e., integrated time between two MCS is pde_its*dt.
Definition of initial condition			
n_init_cells	-	1	Number of cells in initial condition
size_init_cells	-	50	Number of Eden growth [39] cycles used to generate initial cells from single pixels
sizex	-	200	Width of CPM and PDE field in pixels.
sizey	-	200	Height of CPM and PDE field in pixels.
divisions	-	7	Number of divisions of initial cell prior to start of simulation (used to generate a spheroid of cells).
mcs	-	1000000	Total number of Monte Carlo step in simulation.
rseed	-	-1	Random seed. rseed=-1 means random seed is generated from current time in milliseconds (note: do not use this option on clusters as many seeds may be generated within the same millisecond).
subfield	-	1.0	Initial cells are spread over rectangle of xsize/subfield by ysize/subfield centered in the total domain.

`relaxation`	-	`0`	Indicates the number of MCS after which the PDE component is started. This allows one to relax the CPM initial condition (e.g. allow cells to round up).
Parameters defining the simulation output			
`storage_stride`	-	`100`	Gives the number of MCS before a next image or measurement is written.
`graphics`	-	`true`	TST displays graphics on screen if set to 'true'. If set to 'false' program runs on background.
`store`	-	`false`	TST writes output to files if set to 'true' (as defined in the main simulation file).
`datadir`	-	`.`	Output folder for output data.